Fundusze Europejskie dla Wielkopolski
Dofinansowane przez Unię Europejską
SAMORZĄD WOJEWÓDZTWA WIELKOPOLSKIEGO

Analiza danych i uczenie maszynowe

Materiały dydaktyczne dla uczestników szkolenia

Paweł Sobczak

Konin 2026

Tytuł

Analiza danych i uczenie maszynowe

Materiały dydaktyczne dla uczestników szkolenia

Autor

Paweł Sobczak

Projekt pn.
„Rozwój studiów o profilu praktycznym i form kształcenia ustawicznego dostosowanych do potrzeb Wielkopolski Wschodniej”,
realizowany przez Akademię Nauk Stosowanych w Koninie,
jest współfinansowany przez Unię Europejską
ze środków Funduszu na rzecz Sprawiedliwiej Transformacji
w ramach programu Fundusze Europejskie dla Wielkopolski 2021-2027

Fundusze Europejskie dla Wielkopolski
Dofinansowane przez Unię Europejską
SAMORZĄD WOJEWÓDZTWA WIELKOPOLSKIEGO

ANS W KONINIE

Wydawca
Akademia Nauk Stosowanych w Koninie
ul. Przyjaźni 1, 62-510 Konin

Spis treści

Wprowadzenie

Do nauki analizy danych i uczenia maszynowego można wykorzystać różne języki programowania, jednak w praktyce warto wybierać technologie, które są szeroko stosowane na rynku, posiadają bogate zaplecze materiałów edukacyjnych i przykładów, a jednocześnie charakteryzują się niskim progiem wejścia. Oznacza to, że już przy podstawowej wiedzy możliwe jest samodzielne wykonywanie analiz oraz budowanie pierwszych modeli.

Językiem, który w największym stopniu spełnia te wymagania, jest Python. Jest to nowoczesny i uniwersalny język programowania o otwartym kodzie źródłowym, wspierający różne paradygmaty programowania, w tym podejście proceduralne, obiektowe oraz funkcyjne. Python należy obecnie do najczęściej wykorzystywanych języków programowania na świecie, a jego popularność w obszarze analizy danych i uczenia maszynowego wynika z bardzo rozbudowanego ekosystemu bibliotek oraz narzędzi analitycznych.

Python jest językiem wieloplatformowym – oprogramowanie w nim napisane może działać na najczęściej używanych systemach operacyjnych, takich jak Windows, Linux oraz macOS, a także współpracować z innymi środowiskami i technologiami. Dzięki temu znajduje on zastosowanie zarówno w środowiskach akademickich, jak i w projektach komercyjnych.

W kontekście analizy danych i uczenia maszynowego Python oferuje biblioteki umożliwiające:

- ✓ efektywne przetwarzanie i analizę danych numerycznych i tabelarycznych (np. NumPy, Pandas),
- ✓ wizualizację danych i wyników analiz (np. Matplotlib, Seaborn),
- ✓ budowę, trenowanie oraz ocenę modeli uczenia maszynowego (np. scikit-learn),
- ✓ tworzenie bardziej zaawansowanych modeli sztucznej inteligencji i uczenia głębokiego (np. TensorFlow, PyTorch).

Choć Python jest głównym językiem wykorzystywanym w trakcie kursu, warto zaznaczyć, że w praktyce analitycznej często współpracuje on z innymi technologiami, w szczególności z językiem SQL, wykorzystywanym do pracy z bazami danych, oraz narzędziami statystycznymi i systemami raportowymi.

Warto również zaznaczyć, że w analizie danych i uczeniu maszynowym wykorzystywane są także inne języki programowania, w szczególności R, który znajduje zastosowanie głównie w analizach statystycznych, badaniach naukowych oraz zaawansowanym wnioskowaniu statystycznym. W praktyce rynkowej Python i R często się

uzupełniają, jednak w ramach niniejszego kursu Python został wybrany jako język wiodący ze względu na swoją uniwersalność oraz szerokie zastosowanie w projektach analitycznych i komercyjnych.

Ze względów organizacyjnych kurs koncentruje się na podstawowych, ale kluczowych zagadnieniach związanych z analizą danych i uczeniem maszynowym w języku Python. Zakres materiału został dobrany w taki sposób, aby umożliwić uczestnikom samodzielną pracę z danymi, wykonywanie analiz oraz budowę prostych modeli predykcyjnych.

Skrypt należy traktować jako praktyczne wprowadzenie do świata analizy danych i uczenia maszynowego, z ograniczoną ilością teorii oraz dużą liczbą przykładów i ćwiczeń, które pozwalają stopniowo rozwijać umiejętności analityczne i programistyczne.

Aby nauczyć się analizy danych i uczenia maszynowego, nie wystarczy tylko czytać – kluczowe jest samodzielne pisanie kodu, praca z danymi oraz wielokrotne eksperymentowanie z modelami.

Skrypt został podzielony na dwa zasadnicze rozdziały, które odpowiadają kolejnym etapom nauki i stopniowo wprowadzają uczestnika w obszar analizy danych oraz uczenia maszynowego.

Pierwszy rozdział stanowi wprowadzenie do języka Python. Jego celem jest zapoznanie uczestników z podstawami programowania, składnią języka oraz najważniejszymi konstrukcjami, które będą wykorzystywane w dalszej części kursu. Zakres materiału został dobrany w taki sposób, aby umożliwić swobodne posługiwanie się językiem Python jako narzędziem do pracy z danymi.

Drugi rozdział poświęcony jest praktycznej drodze od analizy danych do uczenia maszynowego. W tej części uczestnicy uczą się, jak pozyskiwać, przetwarzać i analizować dane, jak je wizualizować oraz w jaki sposób budować i oceniać modele uczenia maszynowego. Rozdział ten koncentruje się na rzeczywistym procesie pracy analityka i specjalisty ML, obejmując pełny cykl pracy z danymi – od surowych danych do wniosków i predykcji. Taki podział skryptu pozwala na systematyczne rozwijanie kompetencji, łącząc naukę podstaw programowania z praktycznym zastosowaniem Pythona w analizie danych i uczeniu maszynowym.

Python

Przygotowanie środowiska do pracy z analizą danych i uczeniem maszynowym

Zanim rozpoczniemy naukę analizy danych i uczenia maszynowego oraz pisanie własnych programów, konieczne jest przygotowanie odpowiedniego środowiska pracy. W trakcie kursu wykorzystywane będą narzędzia umożliwiające zarówno klasyczne programowanie, jak i interaktywną analizę danych.

1) Instalacja języka Python

Pierwszym krokiem jest instalacja aktualnej wersji języka Python. Oficjalną wersję instalacyjną należy pobrać ze strony: https://www.python.org/downloads/.

Podczas instalacji zaleca się zaznaczenie opcji **Add Python to PATH**, co umożliwi uruchamianie Pythona z poziomu wiersza poleceń oraz integrację z innymi narzędziami. Pliki z kodem programu w języku Python zapisywane są z rozszerzeniem **.py** (np. analiza_danych.py).

2) PyCharm – środowisko IDE do programowania w Pythonie

PyCharm (wersja Community) to rozbudowane środowisko programistyczne (IDE), szczególnie przydatne przy:

- pisaniu większych programów i projektów,
- tworzeniu aplikacji,
- pracy z kodem w sposób uporządkowany i modułowy.

PyCharm oferuje m.in.:

- podpowiedzi składni i automatyczne uzupełnianie kodu,
- wykrywanie błędów na etapie pisania programu,
- zarządzanie bibliotekami i środowiskami wirtualnymi,
- wygodne uruchamianie i debugowanie programów.

Środowisko PyCharm można pobrać ze strony: https://www.jetbrains.com/pycharm/download/

Uruchomienie programu w PyCharm odbywa się:

- za pomocą przycisku **Run**,
- lub skrótu klawiszowego **Shift + F10**.

PyCharm jest szczególnie polecany do nauki podstaw programowania oraz pracy nad projektami, gdzie kod zapisany jest w plikach **.py**.

3) Jupyter Notebook – interaktywna analiza danych

Jupyter Notebook to narzędzie stworzone z myślą o analizie danych i uczeniu maszynowym. Umożliwia pracę w formie tzw. notebooków, które składają się z komórek zawierających:

- kod Pythona,
- tekst opisowy (Markdown),
- wyniki obliczeń,
- wykresy i wizualizacje.

Jupyter Notebook jest szczególnie przydatny do:

- eksploracyjnej analizy danych,
- testowania fragmentów kodu,
- prezentowania wyników analiz krok po kroku,
- nauki i eksperymentowania z modelami ML.

Kod w notebooku uruchamia się komórka po komórce, co pozwala na bieżąco obserwować wyniki działania programu. Jupyter pracuje na plikach **.ipynb**.

Aby zainstalować Jupyter wystarczy w konsoli **cmd** wpisać polecenie: **pip install jupyter**.

Jak uruchomić Jupyter Notebook:

- za pomocą konsoli **cmd** przejść do folderu, gdzie znajdują się pliku projektu i wykonać polecenie: **jupyter notebook**,
- lub w przypadku nowszego interfejsu należy w **cmd** wpisać: **jupyter lab**.

4) Google Colab – Python i Jupyter w przeglądarce

Google Colab (Colaboratory) to internetowa wersja środowiska Jupyter Notebook, działająca bez konieczności instalacji oprogramowania na komputerze. Colab pracuje na plikach **.ipynb**.

Zalety Google Colab:

- działa bezpośrednio w przeglądarce,
- nie wymaga instalacji Pythona ani bibliotek,
- umożliwia zapisywanie notebooków na Dysku Google,
- pozwala korzystać z dodatkowych zasobów obliczeniowych (CPU/GPU).

Google Colab jest idealnym narzędziem:

- na początek nauki,
- do szybkiego testowania kodu,
- do pracy na różnych komputerach,
- do wspólnej pracy i udostępniania analiz.

Dostęp do Google Colab: https://colab.research.google.com/.

Kiedy używać którego narzędzia?

- **PyCharm** – gdy tworzymy klasyczne programy i projekty w plikach .**py**, uczymy się struktury kodu i pracy programistycznej.
- **Jupyter Notebook** – gdy analizujemy dane, tworzymy wykresy i eksperymentujemy z modelami.
- **Google Colab** – gdy chcemy pracować bez instalacji lub szybko uruchomić analizę w przeglądarce.

Tak przygotowane środowisko umożliwia płynne przejście od nauki programowania, przez analizę danych, aż po budowę modeli uczenia maszynowego.

Komentarz

Bardzo często podczas pisania programów komputerowych komentuje się pewne fragmenty kodu lub pewne linie w celu sprawdzenia innego kodu, gdy nie chcemy usuwać poprzedniej wersji lub chcemy wyjaśnić pewne zapisy w kodzie. Kod programu opatrzony komentarzem nie jest interpretowany przez kompilator. Stosowanie komentarzy w kodzie jest dobrą praktyką i ułatwia zrozumienie kody nawet po czasie lub przez innego programistę. Wyróżniamy **komentarz jednoliniowy**: jeżeli linie tekstu poprzedzimy znakiem # oraz **komentarz blokowy**, gdy tekst znajduje się w *”””* *”””*. Przykład użycia komentarza przedstawia Rys. 1.

1.py

```
1
2  # przykładowy komentarz (gdy chcemy zakomentować jedną linie)
3
4  """
5  komentarz blokowy
6  (gdy chcemy zakomentować
7  kilka lini tekstu)
8  """
9
```

Rys. 1. Komentarz jednoliniowy i blokowy w Pythonie.

Źródło: Opracowanie Własne

Zmienne

Pierwszym ważnym elementem w programowaniu są zmienne, czyli nazwy (pudełka), które potrafią przechowywać wartości różnego typu. Operowanie na zmiennych to jedna z najważniejszych funkcjonalności, jakie oferują języki programowania. Krótko można napisać, że zmienna to po prostu nazwa, która wskazuje na jakąś wartość (przechowuje jakąś wartość). Język programowania Python posiada kilka wbudowanych typów danych dla zmiennych, takich jak: liczby całkowite, rzeczywiste itp. W Python podczas tworzenia (deklarowania) zmiennej nie musisz podawać jaki typ danych będziesz przechowywać w zmiennej. Po prostu podaje się nazwę zmiennej i przypisuje się jej wartość, np. liczba_km = 20.5. Znak równości (=) to **operator przypisania**. Na Rys. 2. zostały pokazane przykładowe deklaracje zmiennych. Wartości zmiennych mogę się zmieniać w czasie działania programu.

2.py

```
1  zmienna = 5
2  moje_imie = "PAWEł"
3  nazwaJezyka = "Python"
4  liczbaOsob = 10
5  srednia_ocen = 4.85
6
```

Rys. 2. Deklaracja zmiennych w języku Python.

Źródło: Opracowanie Własne

Klika zasad tworzenia zmiennych:

- nazwy zmiennych mogą być dowolnie długie,
- mogą zawierać zarówno litery, jak i liczby, ale nie mogą rozpoczynać się od liczby,
- choć dozwolone jest użycie wielkich liter, w przypadku nazw zmiennych wygodne jest stosowanie wyłącznie małych liter,
- w przypadku nazw złożonych zalecane jest stosowanie znaku podkreślenia _ lub metody camelCase (np. ile_osob_w_sklepie, liczbaOsob),
- takie same nazwy, ale napisane małymi bądź wielkimi literami, oznaczają różne zmienne,
- nazwy zmiennych nie mogą zawierać spacji,
- zmiennym nie można nadawać nazw zastrzeżonych dla instrukcji języka Python (np. and, for, if, del, while, …itp.).

Instrukcja print

Instrukcja *print* pozwala wyświetlać na ekranie tekst lub wartości zmiennych. W ogólności można zapisać *print(wartość)*, gdzie wartością może być tekst ”Programowanie w Python” lub wartość zmiennej. Przykład użycia instrukcji *print* przedstawia Rys. 3. W przykładzie zadeklarowano dwie zmienne, w pierwszej kolejności zostaje wyświetlony tekst ”Programowanie w Python”, a następnie zostają wyświetlone wartości zmiennych *zmienna* i *moje_imie*. W instrukcji *print* tekst zasadniczo powinno się zapisywać w ” ”.

3.py

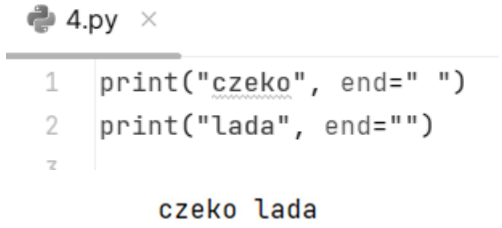
```
1  zmienna = 5
2  moje_imie = "PAWEŁ"
3
4  print("Programowanie w Python")
5  print(zmienna)
6  print(moje_imie)
7
```

```
Programowanie w Python
5
PAWEŁ
```

Rys. 3. Instrukcja *print* (kod programu i wynik uruchomienia).

Źródło: Opracowanie Własne

Instrukcja *print* po każdym wyświetleniu wartości (tekstu lub wartości zmiennych) przechodzi do nowej linii. Czasami jednak pożądane jest pozostać w bieżącej linii, wówczas znak nowej linii można zastąpić na inny znak, np. spację: *print(wartość, end=" ")*. Rys. 4. przedstawia przykład zamiany znaku nowej linii na znak spacji w instrukcji *print*. *Print* może również wyświetlić wynik instrukcji (operacji) oraz kilka zmiennych na raz jak zostało to pokazane na Rys. 5. W tym przykładzie pokazany jest wynik odejmowania zmiennych oraz połączenie napisów z wartością zmiennej. Tych zmiennych w instrukcji *print* może być kilka, a nawet kilkanaście. Można również zastosować odpowiednie formatowanie wyświetlanych wartości zmiennych, jednak nie będzie to przedmiotem tego skryptu.



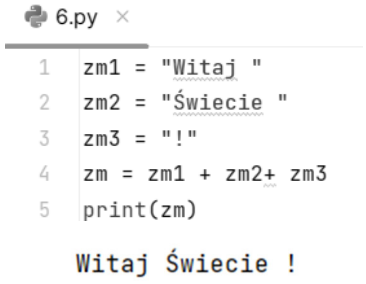
Rys. 4. Instrukcja *print*, zamiana znaku nowej linii na spację.

Źródło: Opracowanie Własne

Rys. 5. Wykorzystanie instrukcji *print*.

Źródło: Opracowanie Własne

W przypadku zmiennych tekstowych tzw. łańcuchów można dokonywać konkatenacji, czyli łączenia kilku zmiennych tekstowych w jeden. Operator + łączy dwa łańcuchy w jedną całość, tworząc nowy, dłuższy łańcuch jak zostało to pokazane na Rys. 6.



Rys. 6. Konkatenacja łańcuchów.

Źródło: Opracowanie Własne

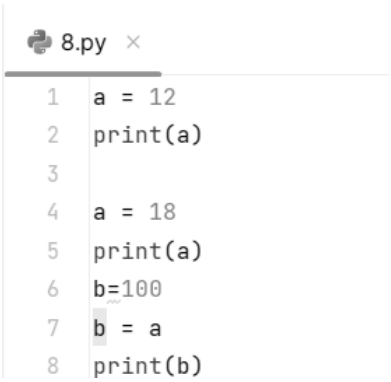
Bardzo często zdążają się sytuacje, że w jednej instrukcji *print* trzeba wyświetlić zmienne różnych typów, wówczas bardzo pomocna jest notacja *f-string*, która w łatwy sposób pozwala wyświetlić zmienne różnego typu w połączeniu z tekstem. Przykład notacji *f-string* przedstawia Rys. 7. Stosując powyższą notację przed cudzysłów zawierający tekst wstawia się literę *f*, czyli *f"tekst"*, a poszczególne zmienne w tekście umieszcza się w nawiasach klamrowych *{}*, taki zapis w prosty sposób pozwala wyświetlić w instrukcji *print* zmienne różnych typów. Modyfikacji wartości zmiennej wykonujemy podobnie, jak byśmy przypisywali wartość do zmiennej z tą różnicą, że musimy podać nazwę istniejącej już zmiennej, aby ją zmodyfikować. Możemy również zmiennej przypisać wartość innej zmiennej, czy też nadpisać wartość innej zmiennej.

nazwaStarejZmiennej = nowaWartośćStarejZmiennej

Rys. 7. Notacja *f-string*, czyli wyświetlanie zmiennych różnych typów w połączeniu z tekstem

Źródło: Opracowanie Własne

Przykład modyfikacji zmiennej i przypisanie zmiennej wartości innej zmiennej przedstawia Rys. 8.



Rys. 8. Zmiana wartości zmiennych

Źródło: Opracowanie Własne

Na ekranie zobaczymy wartości 12, 18, 18, ponieważ na początku zmienna *a* miała wartość 12, następnie wartość zmiennej *a* została zmodyfikowana na wartość 18. Zmienna *b* miała wartość 100, jednak w linii siódmej do *b* została przypisana wartość zmiennej *a*, czyli 18.

Podstawowe typy zmiennych

Jak już zostało wspomniane język programowania Python nie wymaga podawania typu zmiennej podczas jej deklaracji, jednak możemy wskazać podstawowe typy zmiennych w zależności od przechowywanych przez nie wartości, tj.: **liczby** (int, float), **tekst** (str) i **typ logiczny** (bool). W języku tym za pomocą instrukcji *type* można sprawdzić typ zamiennej lub wartości

type(nazwaZmiennej).

Rys. 9. prezentuje, jak sprawdzić typ zmiennej lub wartości. *Int* to zmienne typu całkowitego (pozwalają przechowywać liczby całkowite), *float* to zmienne typu zmiennoprzecinkowego (pozwalają przechowywać liczby rzeczywiste, w których rozdziela się część całkowitą od dziesiętnej za pomocą kropki .). Nic nie stoi na przeszkodzie, by zmienną przekonwertować z typu rzeczywistego na całkowity i odwrotnie. Możemy również liczbę wprowadzoną w postaci napisu przekonwertować na zmienną typu całkowitego lub rzeczywistego, więcej na ten temat będzie poruszone podasz wprowadzania wartości z klawiatury (konsoli). Funkcja *int()*

konwertuje zmienną do tupu całkowitego, natomiast funkcja *float()* do typu rzeczywistego. Przykładowe konwersje zmiennych przedstawia Rys. 10.

9.py

```
1
2  a = 0.5
3  b = 2
4  c = "Cześć"
5  print (type(a))
6  print (type(b))
7  print (type(c))
8  print (type(100))
9  print (type("Python"))
```

```
<class 'float'>
<class 'int'>
<class 'str'>
<class 'int'>
<class 'str'>
```

Rys. 9. Sprawdzenie typu zmiennej i wartości

Źródło: Opracowanie Własne

Zmienna *d* na początku ma wartość 3.4, jednak po konwersji do typu całkowitego ma wartość 3. Z kolei zmienna *e* to wartość zmiennej *b* przekonwertowana do typu rzeczywistego, czyli 2.0. Wartość zmiennej *f*, to wartość 1.5, gdyż napis został przekonwertowany do wartości rzeczywistej, a linia 11 spowoduje wyświetlenia wartości 4.5 w konsoli. Język Python bardzo dobrze radzi sobie z konwersją różnych typów zmiennych.

10.py

```
1  a = 1.4
2  b = 2
3  c = 2.4
4  d = a + b          # 1.4 + 2 = 3.4
5  print(d)
6  d = int(d)         # 3.4 do int, to 3
7  e = float(b)       # 2 do float, to 2.0
8  f = float("1.5") # napis "1.5" do float, to 1.5
9  print (d)
10 print (e)
11 print(f+3.0)       # 1.5 + 3.0 = 4.5
```

```
3.4
3
2.0
4.5
```

Rys. 10. Konwersja zmiennych

Źródło: Opracowanie Własne

Zmienne tekstowe (str)

Typ *str* reprezentuje w Pythonie wszelkiego rodzaju teksty/napisy. Tekst możemy umieścić pomiędzy znakami pojedynczego (' ') lub podwójnego (" ") cudzysłowu – efekt będzie dokładnie taki sam, np. imie = "Paweł", jezyk = 'Python'. Zadeklarowaliśmy dwie zmienne tekstowe. Również wartości zmiennych typu całkowitego lub rzeczywistego można konwertować do wartości tekstowej za pomocą funkcji *str()*, jak zostało to pokazane na Rys. 11. W linii drugiej wiek został skonwertowany do napisu.

Rys. 11. Konwersja zmiennej liczbowej do tekstowej

Źródło: Opracowanie Własne

Zmienne logiczne (bool)

Wartości logiczne reprezentują logiczną prawdę lub fałsz. W języku programowania Python istnieją predefiniowane nazwy odpowiednio *True* dla logicznej prawdy (1) i *False* dla logicznego fałszu (0):

```
p = True      # deklaracja wartości logicznej "prawda"
f = False     # deklaracja wartości logicznej "fałsz".
```

W celu przekształcenia dowolnej wartości na wartość logiczną można wykorzystać funkcję wbudowaną *bool()*. Dla przykładu:

```
a = 1
b = 0
bool(a)      # True
bool(b)      # False.
```

Konwersja wartości zmiennej *a* zwróci *True*, czyli prawdę, a zmiennej *b* zwróci *False*, czyli fałsz.

Operatory arytmetyczne

Żadne obliczenia w programie nie byłyby możliwe, gdyby nie operatory arytmetyczne. Wyróżniamy następujące operatory arytmetyczne (+, -, *, /, **%**, **).

Dodawanie +	(2 + 4)
Odejmowanie -	(11 - 5)
Mnożenie *	(2 * 4)
Dzielenie /	(6 / 2)
Modulo – reszta z dzielenia **%**	(10 % 3)
Potęgowanie **	(2 ** 2)

Na Rys. 12. przedstawiono użycie podstawowych operatorów arytmetycznych (+, -, * i /). Póki co, kod programu nie został zabezpieczony na ewentualność dzielnie przez 0, czyli żeby *liczba2* była różna od zera. Takie zabezpieczenie zrobimy po wprowadzeniu instrukcji warunkowej *if* i operatorów relacji. Warto już zaznaczyć, że w linii 2 i 3 kodu do liczb *liczba1* i *liczba2* została przypisana wartość wczytana z klawiatury, które zostały skonwertowane do typu liczbowego zmiennoprzecinkowego *float*, o czym będzie mowa w dalszej części rozdziału (interaktywny program).

12.py

```
1  print("Prosty kalkulator")
2  liczba1 = float(input("Podaj pierwszą liczbę: "))
3  liczba2 = float(input("Podaj drugą liczbę: "))
4  print("Wynik operacji na liczbach:")
5  print(f"{liczba1} + {liczba2} = {liczba1+liczba2}")
6  print(f"{liczba1} - {liczba2} = {liczba1-liczba2}")
7  print(f"{liczba1} * {liczba2} = {liczba1*liczba2}")
8  print(f"{liczba1} / {liczba2} = {liczba1/liczba2}")
```

```
Prosty kalkulator
Podaj pierwszą liczbę: 3
Podaj drugą liczbę: 2
Wynik operacji na liczbach:
3.0 + 2.0 = 5.0
3.0 - 2.0 = 1.0
3.0 * 2.0 = 6.0
3.0 / 2.0 = 1.5
```

Rys. 12. Operatory arytmetyczne (+, -, *, /)

Źródło: Opracowanie Własne

Operatory relacji

Dostępne operatory relacji, które wykorzystuje się w instrukcji warunkowej *if* oraz pętlach są zebrane poniżej. Wynikiem porównania jest wartość logiczna *True* dla prawdy lub

False dla fałszu. Przykłady użycia operatorów relacji zostaną pokazane po wprowadzeniu wspomnianej wyżej instrukcji *if*.

> większy niż
< mniejszy niż
== równy względem
>= większy lub równy względem
<= mniejszy lub równy względem
!= różny względem

Operatory logiczne

W języku programowania Python wykorzystuje się również operatory logiczne, podobnie jak operatory relacji najczęściej w połączeniu z instrukcją warunkową *if* oraz w pętlach. Operator *AND*, to w logice „*i*" (koniunkcja), natomiast *OR*, to w logice „*lub*" (alternatywa), z kolei *NOT*, to zaprzeczenie (negacja). Gdy zanegujemy prawdę *True* to otrzymamy fałsz, czyli *False* i na odwrót. Logiczne „*lub*" zwraca prawdę, gdy przynajmniej jeden z warunków jest prawdziwy, w przeciwnym razie zwraca fałsz. Z kolei logiczne „*i*" zwraca prawdę w przypadku, gdy wszystkie warunki są prawdziwe, w przeciwnym razie zwraca fałsz.

AND
OR
NOT

Interaktywny program

Interaktywny program, to program, w którym użytkownik ma możliwość wprowadzania danych w konsoli w czasie rzeczywistym. Python udostępnia wbudowaną funkcję *input*, która umożliwia wprowadzenie danych przez użytkownika. Po wywołaniu funkcji *input* w konsoli pokazuje się kursor oczekiwania na wprowadzenie ciągu znaków. Po wprowadzaniu wartości (ciągu znaków) użytkownik zatwierdza wprowadzoną wartość klawiszem *Enter*. Wprowadzony ciąg znaków z klawiatury musi zostać przypisany do jakieś zmiennej. Warto zaznaczyć bardzo ważną rzecz: jeżeli nawet wczytamy znaki będące liczbą, to należy pamiętać,

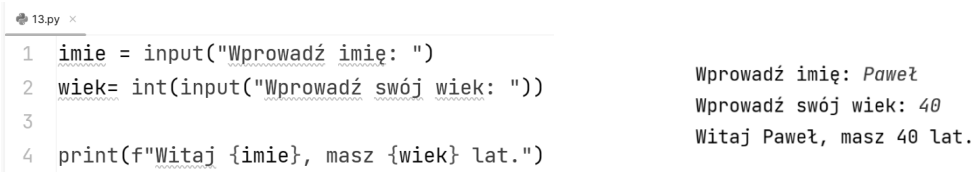
że to nie jest liczba, tylko ciąg znaków (string), który należy przekonwertować do wartości liczbowej.

imie = input("Wprowadź imię: ")

Po wywołaniu funkcji *input* na ekranie pokazuje się napis *„Wprowadź imię":* oraz kursor oczekiwania na wprowadzenie ciągu znaków. Użytkownik wprowadza imię (ciąg znaków), który zatwierdza klawiszem *Enter*, wprowadzony ciąg znaków zostaje przypisany do zmiennej *imie* (*string* jest zwracany poprzez funkcję *input* i przypisany do zmiennej *imie*). Poniżej podobnie wprowadzany jest ciąg znaków *wiek*, który ma być docelowo wartością liczbową, dlatego dodatkowo jest konwertowany do typu *int*, czyli wartości liczbowej całkowitej. Gdy chcemy dokonać konwersji ciągu znaków do wartości zmiennoprzecinkowej używamy funkcji *float*.

wiek= int(input("Wprowadź swój wiek: "))

Omawiane wyżej dwa przykłady zostały pokazane Rys. 13.



Rys. 13. Interaktywny program, czyli wprowadzanie ciągów znaków z konsoli.

Źródło: Opracowanie Własne

Typy sekwencyjne

Sekwencyjne typy danych służą do zapamiętywania wielu wartości w pojedynczej zmiennej, w odróżnieniu od typów prostych, takich jak *int*, *float*, które w pojedynczej zmiennej mogą zachować tylko jedną wartość.

Typ napisowy

Tak naprawdę napisy są sekwencjami znaków. Każdy typ sekwencyjny pozwala na dostęp do każdego swojego elementu z osobna. Aby uzyskać dostęp do znaku na określonej pozycji podajemy nazwę zmiennej oraz indeks (numer porządkowy liczony od lewej, zero oznacza pierwszy znak napisu) w nawiasach kwadratowych:

imie = ”Paweł”

0	1	2	3	4
P	a	w	e	ł

imie[1] *# ‘a’*
print(imie[4]) *# wyświetli znak ł na ekranie konsoli.*

Należy zapamiętać, że znaki (elementy) są numerowane od 0, czyli powyższy napis *imie* składa się z 5 elementów numerowanych od 0 do 4. Można również numerować elementy liczbami ujemnymi, jednak celowo nie wprowadzam tego sposobu indeksowania, by nie zmniejszać czytelności indeksowania typów sekwencyjnych. Aby poznać długość napisu (liczbę elementów), posługujemy się funkcją *len*:

len(imie) ***# 5, numerowane od 0 do 4.***

Przykład odwołania się do pojedynczych znaków napisu oraz długość napisu prezentuje Rys. 14.

14.py

```
#Typy sekwencyjne: typ napisowy
imie = "Paweł"
nazwisko = "Sobczak"

print(imie)
print(f"Liczba elemntów zmiennej imie: {len(imie)}")
print(imie[4])
print(nazwisko[0])
```

```
Paweł
Liczba elemntów zmiennej imie: 5
ł
S
```

Rys. 14. Typ napisowy (odwołalnie do pojedynczych znaków, długość napisu).

Źródło: Opracowanie Własne

Dla typu napisowego istnieje szereg przydanych metod, które bardzo ułatwiają pracę programiście, poniżej zostały zaprezentowane przykładowe metody. Jeżeli nasza zmienna napisowa to *imie = "Pawel"*, to możemy dla niej uruchomić pewne metody:

imie.capitalize() - zmienia pierwszą literę na dużą
imie.count(Pa) - zlicza wystąpienie podciągu Pa w napisie imie
imie.isdigit() - sprawdza czy wszystkie znaki są cyframi
imie.islower() - sprawdza czy wszystkie litery są małe
imie.isupper() - sprawdza czy wszystkie litery są duże
imie.replace(old, new) - zastępuje stary podciąg nowym
imie.strip() - usuwa początkowe i końcowe białe znaki.

Warto zaznaczyć, że uruchomienie metody dla zmiennej znakowej odbywa się przez podanie nazwy zmiennej, następnie znaku kroki (.) i nazwy metody. Wynika to z programowania obiektowego, które poznacie Państwo na szkoleniu.

nazwa_zmiennej.nazwa_metody

Listy

Najpopularniejszym i najczęściej stosowanym typem zmiennych zawierającym więcej niż jedną wartość są listy (od angielskiego „list"), czasami nazywane też tablicami. Lista to uporządkowany zbiór różnych elementów. Najczęściej wewnątrz listy stosuje się jeden typ zmiennych, ale nic nie stoi na przeszkodzie, aby w jednej liście umieścić wartości zupełnie różnych typów (Python na to pozwala, jednak język C/C++ już nie). Zmienne tworzymy, zapisując pomiędzy nawiasami kwadratowymi („[" i „]") elementy, które chcemy, aby nasza lista przetrzymywała. Poniżej została zadeklarowana *lista1* złożona z 5 elementów numerowanych od 0 do 4 zawierająca zmienne różnego typu.

lista1 = [5, 1.28, "Programowanie", "Python", -2.36]

indeks	0	1	2	3	4
wartość	5	1.28	Programowanie	Python	-2.36

Jak już zostało to wspominane, listy najczęściej są złożone z elementów tego samego typu. Poniżej została zaimplementowana lista złożona z liczb całkowity oraz pusta lista, która nie zawiera na ten moment żadnego elementu.

```
lista1 = [0, 2, 4, 6, 8]
lista2 = []
```

lista1 przechowuje 5 elementów, które zostały umieszczone w nawiasach kwadratowych[]. Można wyświetlić wszystkie elementy listy lub pojedynczy element. Można również dokonać zmiany wartości elementu w liście.

```
print(lista1)       # [0, 2, 4, 6, 8]
print(lista1[2])    # 4
lista1[2] = 5        # element o indeksie 2 będzie miał wartość 5
print(lista1)       # [0, 2, 5, 6, 8]
```

Indeksowanie list jest identyczne jak indeksowanie typu napisowego i zaczyna się od 0. Również w innych językach programowania np. C/C++ tablice indeksuje się od 0. Nasza *lista1* składa się z 5 elementów indeksowanych od 0 do 4, gdzie pierwszy element to 0, a ostatni (4) wartość 8. Nie ma indeksu o wartości 5! Jest to bardzo ważne przy pracy z listami, by nie wyjść poza zakres listy.

- ✓ **Dodawanie elementu do *listy***

Aby dodać element do listy, używamy funkcji *append*, której jako parametr podajemy wartość, jaką chcemy dodać do naszej listy, np.

```
print(lista1)          # [0, 2, 5, 6, 8]
lista1.append(10)      # dodanie elementu 10 do listy
print(lista1)          # [0, 2, 5, 6, 8, 10].
```

Czyli do wyżej omawianej *lista1* został dodany kolejny element o wartości 10. Element ten został dodany na koniec listy.

- ✓ **Usunięcie elementu po indeksie**

Aby usunąć jakiś element z listy, należy użyć instrukcji *del* od angielskiego słowa *„delete”*, czyli właśnie usuwać. Instrukcja *del* usunie element z listy o określonym indeksie. Należy zwrócić szczególną uwagę na indeks elementu, którego chcemy usunąć, by nie odwoływać się do elementu, który nie istnieje.

```
lista2 = [7, 2, 4, 6, 1]      # [7, 2, 4, 6, 1]
del lista2[2]                 # usuniecie elementu z listy o indeksie 2
print(lista2)                 # [7, 2, 6, 1]
```

Powyżej został utworzona nowa *lista2* złożona z 5 elementów indeksowana od 0 do 4. Następnie został usunięty element o indeksie numer 2, czyli element **4**, po czym zostały wyświetlone elementy listy. Warto zwrócić uwagę, że po takim działaniu nasza lista uległa skróceniu, a więc i numeracja indeksów uległa zmianie. W tej chwili ostatni element tablicy ma numer 3. Użycie indeksu o wartości 4 spowoduje błąd w programie.

- ✓ **Dodanie elementu w dowolne miejsce**

Jeśli chcemy dodać element w konkretne miejsce na liście, musimy znać numer elementu (indeks), przed który chcemy wstawić nową wartość. Dodanie elementu do listy na konkretnej pozycji wykonuje się przez funkcję *insert*, która przyjmuje dwa parametry. Pierwszy parametr funkcji *insert* to *indeks* miejsca, przed które chcemy wstawić nowy element, a drugi to *element*, który chcemy dodać do naszej listy.

```
lista2.insert(2, 4)  # dodanie elementu do listy o wartości 4 na pozycji 2
print(lista2)        # [7, 2, 4, 6, 1]
```

- ✓ **Sprawdzenie czy element występuje na liście**

W celu sprawdzenia czy dany element występuje w liście, stosujemy polecenie:

lista3 = [0, 2, 4, 6, 8] # [0, 2, 4, 6, 8]
print(2 in lista3) # Wyświetli True, bo element 2 jest na liście.

Podobnie jak do typu napisowego istnieje kilka przydatnych metod do obsługi list, przykładowe poniżej:

list(s) konwertuje sekwencję s na listę
s.append(x) dodaje nowy element x na końcu listy s
s.count(x) zlicza wystąpienie x w liście s
s.index(x) zwraca najmniejszy indeks i, gdzie s[i] == x
s.pop(i) zwraca i-ty element z listy i usuwa go z listy s
s.remove(x) odnajduje wartość x i usuwa go z listy s
s.reverse() odwraca w miejscu kolejność elementów listy s.

Przykładowe operacje na liście zostały zebrane na Rys. 15.

15.py

```
lista =["Python", 3, 2.5, "programowanie"]
print(lista)                # wyświetlenie elementów listy
print(lista[0])             # wyświetlenie elementu o indeksie 0
lista1 = [2, 4, 5, 1, 5]
print(lista1)
print(lista1[4])
lista1.append(10)           # dodanie wartości 10 na koniec listy
print(lista1)
del lista1[1]               # usunięcie elementu o indeksie 1
print(lista1)
lista1.insert(2,8)          # wstawienie elementu 8 na pozycję 2
print(lista1)
print(8 in lista1)          # sprawdzenie, czy wartość 8 jest na liście
```

```
['Python', 3, 2.5, 'programowanie']
Python
[2, 4, 5, 1, 5]
5
[2, 4, 5, 1, 5, 10]
[2, 5, 1, 5, 10]
[2, 5, 8, 1, 5, 10]
True
```

Rys. 15. Przykładowe operacje na listach.

Źródło: Opracowanie Własne

Krotki

Innym typem zmiennych, który może przetrzymywać więcej niż jedną wartość, są tak zwane krotki. Krotki są bardzo podobne do list z jedną bardzo ważną różnicą – **dane w krotkach są niezmienne**. Czyli raz utworzona krotka już do końca ma takie elementy, jakie zostały podane przy jej implementacji. Krotki deklaruje się tak samo jak listy, tylko zamiast nawiasów kwadratowych używa się zwykłych ().W przypadku krotek, tak samo jak i w listach,

możemy wyświetlać pojedyncze elementy za pomocą instrukcji *print* lub wszystkie, również indeksujemy je od zera.

```
krotka1 = (1, 3, 5, 7, 9)
print(krotka1)          # (1, 3, 5, 7, 9)
print(krotka1[1])       # wyświetlamy element 1 czyli 3
```

Krotki są bardzo przydatnym typem danych wszędzie tam, gdzie kolejność elementów ma znaczenie, a bardzo nie chcemy, żeby program mógł zmieniać zawartość naszej zmiennej. Zastosowanie krotki może mieć miejsce w zdefiniowaniu parametrów konfiguracyjnych naszego programu np.: połączenie z bazą danych, login do bazy, hasło itp. Przykład wyświetlenia elementów krotki pokazany jest na Rys. 16.

16.py

```
1  krotka = (3, 2, 1, 5)
2  krotka1 = ("Ala", "ma", "kota", 3, 3.05)
3  print(krotka1)
4  print(krotka)            # wyświetlenie wszystkich elementów
5  print(krotka[3])         # wyświetlenie pojednyńczego elementu krotki
```

```
('Ala', 'ma', 'kota', 3, 3.05)
(3, 2, 1, 5)
5
```

Rys. 16. Wyświetlanie elementów krotki.

Źródło: Opracowanie Własne

Zbiory

Trzecim typem zmiennych mogących posiadać więcej niż jedną wartość są zbiory, posiadają one pewną bardzo ciekawą właściwość, która może być bardzo pomocna przy rozwiązywaniu niektórych problemów: **jej elementy nie mogą się powtarzać**. Dodatkową cechą zbiorów jest to, że są nieuporządkowane, a co za tym idzie nie możemy wyświetlać dowolnego ich elementu w taki sposób jak w przypadku list, czy krotek. Można za to dodawać i usuwać elementy ze zbiorów. Zbiory tworzymy za pomocą nawiasów klamrowych {}.

```
zbior1 = {2, 4, 6, 8, 10}     # tworzymy zbiór elementów
print(zbior1)                  # {2, 4, 6, 8, 10}
zbior1.add(1)                  # dodajemy do zbior1 wartość 1
zbior1.add(3) )                # dodajemy do zbior1 wartość 3
```

```
print(zbior1)          # {1, 2, 3,  4, 6, 8, 10}
zbior.remove(10)       # usuwamy ze zbioru wartość 10
print(zbior)           # {1, 2, 3,  4, 6, 8}
zbior2 = set()         # tworzymy zbiór pusty
zbior2= {5, 3, 1}      # przypisujemy lamenty do zbioru
```

Za pomocą metody *add* można dodać elementy do zbioru, z kolei za pomocą *remove* można usunąć element ze zbioru. Funkcja *len* (np. *len*(zbior2)) zwróci liczbę elementów w zbiorze.

Słowniki

Ostatnim z typów danych mogących posiadać więcej niż jedną wartość są słowniki. Słowniki są zupełnie innym typem danych od dotychczas opisywanych. Pierwszy element jest nazywany *kluczem*, a drugi *wartością*. Jednemu elementowi (kluczowi), jest przypisana jakaś wartość. Warto zauważyć, że kolejność elementów w słownikach nie ma znaczenia, ponieważ dane w nich i tak wyszukuje się po kluczu. Słowniki można modyfikować, czyli można do nich dodawać nowe elementy i usuwać już istniejące. Jedyna zasada to taka, że **klucze nie mogą się powtarzać**. W słowniku wszystko może być *kluczem*, tak samo jak i wszystko może być w*artością*. Możliwe są zatem takie przykładowe konstrukcje:

```
na_slowa = {1:'jeden', 2:'dwa', 3:'trzy', 4:'cztery', 5:'pięć'}
print(na_slowa)        # {1:'jeden', 2:'dwa', 3:'trzy', 4:'cztery', 5:'pięć'}

na_cyfry = {'jeden':1, 'dwa':2, 'trzy':3, 'cztery':4, 'pięć':5}
print(na_cyfry)        # {'jeden':1, 'dwa':2, 'trzy':3, 'cztery':4, 'pięć':5}
```

Powyższe konstrukcje pozwalaj zamieniać napisy na liczby i odwrotnie. Słowniki, tak samo jak zbiory, tworzymy przy użyciu nawiasów klamrowych. Pierwszy element to *klucz*, drugi to *wartość*. *Klucz* jest oddzielony od *wartości* dwukropkiem (:), a poszczególne pary *klucz-wartość* przecinakami (,).

```
print(na_slowa[3])       # wyświetli 'trzy'
print(na_cyfry['trzy'])  # wyświetli 3
```

Aby dodać nowy element do słownika, po prostu wpisujemy w nawiasy kwadratowe klucz, którego chcemy użyć i przypisujemy mu wartość:

```
na_slowa[6]=' sześć'        # dodanie do słownika pary o kluczu 6 i wartości 'sześć'
print(na_slowa)             # {1: 'jeden', 2: 'dwa', 3: 'trzy', 4: 'cztery', 5: 'pięć', 6: 'sześć'}
```

Usuwanie elementów ze słownika wygląda podobnie jak w przypadku list, ale zamiast indeksu elementu podajemy klucz, który chcemy usunąć wykorzystując metodę *pop*.

```
na_slowa.pop(3)        # usunięcie elementu ze słownika na_slowa o kluczu 3
print(na_slowa)        # {1: 'jeden', 2: 'dwa', 4: 'cztery', 5: 'pięć'}
na_cyfry.pop('trzy')   # usunięcie elementu ze słownika na_cyfry o kluczu ‘trzy’
print(na_cyfry)        # {'cztery': 4, 'dwa': 2, 'jeden': 1, 'pięć': 5}
```

Instrukcje warunkowe if

Praktycznie prawie w każdym programie są podejmowane pewne decyzje, są pewne warunki, które wpływają na pracę programu. Do podejmowania decyzji w programowaniu służy instrukcja warunkowa *if*, czyli z języka angielskiego „*jeśli*”. Fragment kodu programu wykona się tylko wtedy, gdy będzie spełniony warunek (warunek będzie prawdziwy, czyli będzie miał wartość logiczną *True*). Sytuacji takich jest bardzo dużo, np. dzielnie zostanie wykonane tylko wtedy, gdy dzielnik będzie różny od zera. Instrukcja *if* posiada również opcjonalną, dodatkową część w postaci instrukcji *else*, czyli „*w przeciwnym wypadku*”. Dodatkowa cześć *else* nie jest obowiązkowa, ale bardzo często jest przydatna, gdy chcemy, by program sprawdził jakiś warunek i wykonał jakiś kod, jeśli warunek jest prawdziwy lub wykonał inny kawałek kodu, jeśli warunek był nieprawdziwy (fałszywy). Użycie instrukcji *if – else* wygląda w Pythonie następująco:

```
if (wyrażenie warunkowe):
    instrukcja 1
    instrukcja 2
    ...
```

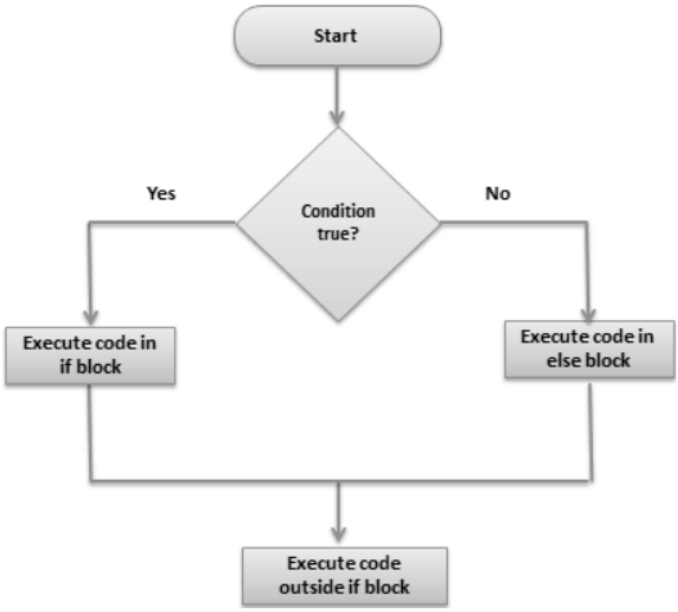
else:

instrukcja 1

instrukcja 2

...

W części „*wyrażenie warunkowe*" wpisujemy, to co chcemy, aby nasz program sprawdził (czyli stawiamy pewien warunek). Wyrażenie warunkowe może być zapisane w nawiasach, jednak nie jest to wymagane. Po części *wyrażenie warunkowe* musimy wpisać dwukropek, co oznacza, że dalej występują instrukcje, które mają być wykonane, jeśli warunek jest prawdziwy. Warto zaznaczyć, że instrukcji może być dowolna ilość, ale wszystkie instrukcje muszą być wcięte względem instrukcji *if*. W ten sposób Python rozpoznaje które instrukcje ma wykonać po sprawdzeniu prawdziwości wyrażenia. Tak samo po instrukcji *else* musimy wstawić dwukropek, a instrukcje muszą być wcięte. Np. w języku programowania *C/C++* wcięcia to tylko dobra praktyka programisty, a operacje (instrukcje) blokuje się za pomocą klamer*{}*. Działanie instrukcji *if* - *else* odzwierciedla Rys. 17.

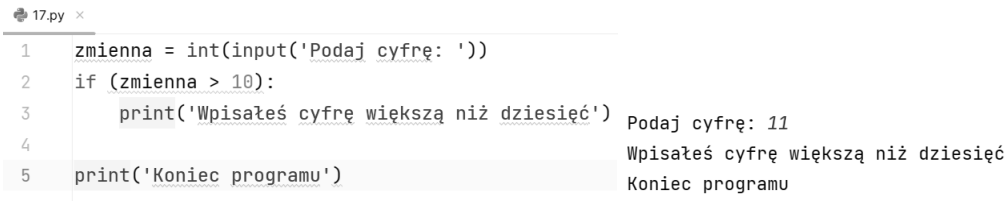


Rys. 17. Instrukcja warunkowa *if* - *else*.

Źródło: Opracowanie własne

Poniżej prosty przykład użycia instrukcji *if* (Rys. 18.). Program wczytuję liczbę z konsoli i konwertuje ją do typu całkowitego. Następnie sprawdzany jest warunek, jeśli

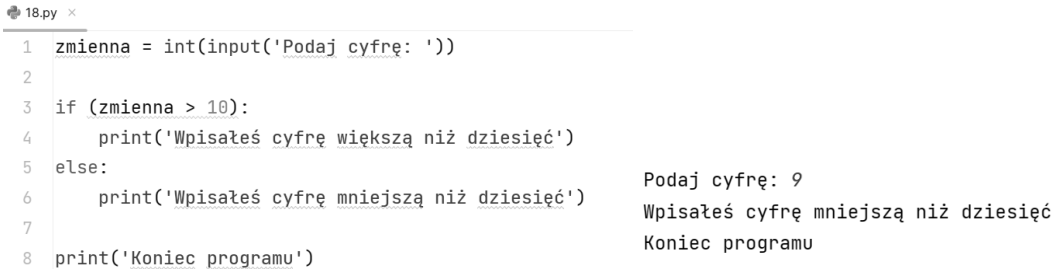
wprowadzona liczba jest większa od 10, wówczas w konsoli wyświetla się komunikat *'Wpisałeś cyfrę większą niż dziesięć'*, gdy jednak mniejsza lub równa 10, to nic się nie dzieje. Kolejną instrukcją jest instrukcja *print* wyświetlająca komunikat *'Koniec programu'* niezależnie od tego, czy warunek był prawdziwy, czy fałszywy.



Rys. 18. Przykład użycia instrukcji warunkowej *if*.

Źródło: Opracowanie własne

Na Rys. 19. została pokazana zmodyfikowana wersja programu z Rys. 18. Modyfikacja polegała na dodaniu dodatkowej cześć *else*, która wykona się, gdy warunek jest nieprawdziwy, tzn., gdy wprowadzona cyfra jest mniejsza od 10. Program ma jeszcze jeden defekt, nieprawidłowo się zachowa, gdy wprowadzimy cyfrę równą 10. W celu usunięcia ww. defektu musimy dodać jeszcze jedną instrukcję *if*, jak zostało to pokazane na Rys. 20.



Rys. 19. Przykład użycia instrukcji warunkowej *if* - *else*.

Źródło: Opracowanie własne

Jak widać z Rys. 20. można zagnieżdżać instrukcje warunkowe, czyli w instrukcji warunkowej umieścić kolejną instrukcję. Linie 5 i 6 kodu z Rys. 20. można połączyć w jedną linie. Zamiast pisać *else:*, a następnie *if()*, można od razu zapisać *elif*, jak zostało to pokazane na Rys. 21.

```
zmienna = int(input('Podaj cyfrę: '))

if (zmienna > 10):
    print('Wpisałeś cyfrę większą niż dziesięć')
else:
    if (zmienna == 10):
        print('Wpisałeś cyfrę równą dziesięć')
    else:
        print('Wpisałeś cyfrę mniejszą niż dziesięć')

print('Koniec programu')
```

```
Podaj cyfrę: 10
Wpisałeś cyfrę równą dziesięć
Koniec programu
```

Rys. 20. Przykład użycia zagnieżdżonej instrukcji warunkowej *if - else*.

Źródło: Opracowanie własne

```
zmienna = int(input('Podaj cyfrę: '))

if (zmienna > 10):
    print('Wpisałeś cyfrę większą niż dziesięć')
elif (zmienna == 10):
        print('Wpisałeś cyfrę równą dziesięć')
else:
        print('Wpisałeś cyfrę mniejszą niż dziesięć')

print('Koniec programu')
```

```
Podaj cyfrę: 9
Wpisałeś cyfrę mniejszą niż dziesięć
Koniec programu
```

Rys. 21. Przykład użycia instrukcji *elif*.

Źródło: Opracowanie własne

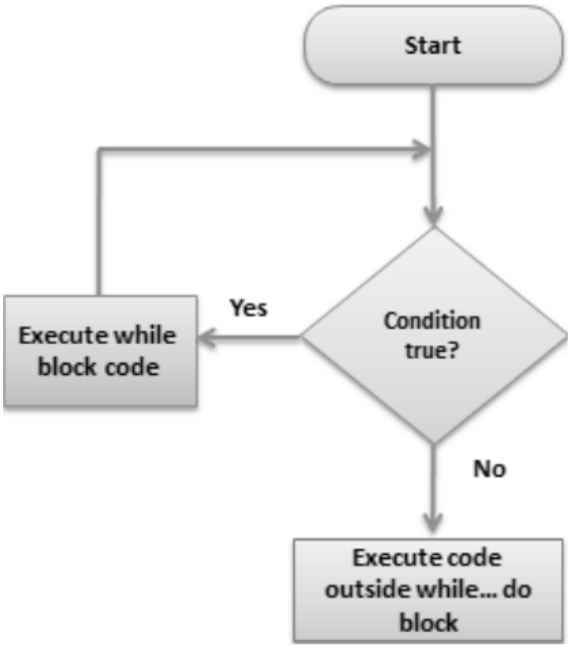
Pętla for

Poznanie instrukcji iteracyjnych (pętli) pozwala programiście rozwiązywać trudniejsze zadnia, problemy. Obok instrukcji warunkowej, znajomość pętli jest konieczna do pisania programów. Pisząc programy, bardzo często zdarzy się, że będziemy chcieli wykonać jakieś zadanie (instrukcję) więcej niż jeden raz. Korzystając z pętli możemy określoną operację (instrukcję) wykonywać z góry określoną ilość razy, np. 1000, 20000, 3 miliony lub tak długo, jak warunek jest prawdziwy. Idę działania instrukcji iteracyjnych (pętli) przedstawia Rys. 22.

Pierwszą pętle jako sobie omówimy, jest pętla *for*. Pętla ta ma kilka postaci. Jako pierwszą opiszemy pętle *for* z zakresem *range*. Pętla ta ma następująco postać:

***for** i **in range**(101):*

<instrukcje>



Rys. 22. Idea działania pętli.

Źródło: Opracowanie własne

W tej konstrukcji, funkcja *range* przyjmuje parametr *i* zwraca kolejno cyfry z zakresu **<0; wartość)**, czyli w naszym wypadku **<0; 101)**. Pisząc prościej pętla wykona się 101 razy, a w każdym obiegu pętli parametr (zmienna) *i* będzie przyjmować odpowiednio wartości: 0, 1, 2, 3, 4, …, 100. Warto zwrócić uwagę, jakie wartość przyjmie parametr *i*. Znak mniejszości **<** oznacza, że zaczynamy od 0 włącznie i kończymy na 101 (czyli wartości zapisanej w nawiasie) ale bez tej wartości, bowiem jest nawias otwarty **)**. Po podaniu wartości funkcji *range*() stawia się **:**, a następnie wypisuje się instrukcje, które mają być wykonane w pętli. Wszystkie instrukcje, które mają być wykonywane w pętli musza być wcięte w stosunku do instrukcji *for*, podobnie jak to było w przypadki instrukcji *if*. Funkcja *range* może przyjmować następujące postacie:

- ✓ bez zakresu początkowego *range*(10) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
- ✓ z zakresem początkowym i końcowy *range*(1,10) # [1, 2, 3, 4, 5, 6, 7, 8, 9],
- ✓ z zakresem początkowym, końcowym i krokiem *range*(1,10,2) # [1, 3, 5, 7, 9].

23.py

```
1  # Pętla for in range
2  for i in range(10):
3      print(i, end=" ")
4
5  print(" ")
6
7  for i in range(1, 8):
8      print(i, end=" ")
9
10 print(" ")
11
12 for i in range(0, 9, 2):
13     print(i, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7
0 2 4 6 8
```

Rys. 23. Przykład użycia pętli *for in range*.

Źródło: Opracowanie własne

W ogólności składnia pętli for jest następująca:

for <*nazwa_zmiennej*> ***in*** <*obiekt*> :
instrukcja1
instrukcja2
instrukcja3

Zamiast <*nazwa_zmiennej*> wstawiamy dowolną nazwę, która będzie wykorzystywana do przechowywania kolejnych elementów pobieranych z <*obiekt*>. Pętle *for* dla przykładu możemy wykorzystać do wypisania wszystkich elementów z listy. Pętla wykona się tyle razy, ile elementów jest na liście. Przykład użycia pętli *for* do odczytu wszystkich elementów listy przedstawia Rys. 24.

24.py

```
1  lista = [0, 2, 4, 8, 10]
2
3  for zm in lista:
4      print (zm, end=" ")
```

```
0 2 4 8 10
```

Rys. 24. Przykład użycia pętli *for* z elementami listy.

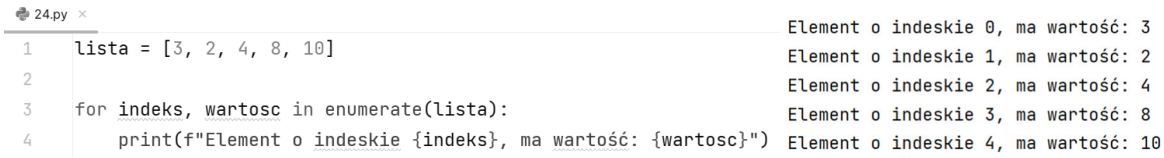
Źródło: Opracowanie własne

W tym przykładzie pętla *for* pobiera kolejne elementy z listy, „wrzuca” je do zmiennej *zm* i następnie przechodzi do wykonywania instrukcji, które jak zawsze są wypisane

po dwukropku i we wcięciu. W tym konkretnym przypadku jest to instrukcja *print*, która wyświetla poszczególne elementy z listy.

Zdarzają się sytuację, że oprócz wyświetlenia elementów listy potrzebujemy wyświetlić indeksy poszczególnych elementów. W takiej sytuacji stosujemy pętle *for* w postaci *enumerate*:

***for** indeks, wartość in **enumerate**(lista):*
print(indeks, wartość)



Rys. 25. Pętla *for* dla list (wyświetlenie indeksów i wartości elementów listy.

Źródło: Opracowanie własne

Warto przypomnieć, że listy numeruje się od 0. Przykład jak odczytywać indeksy i wartości poszczególnych elementów listy za pomocą instrukcji *for* pokazany jest na Rys. 25. Podobnie pętlę *for* można wykorzystać do pracy ze słownikami.

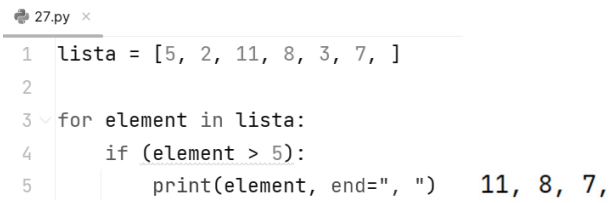
***for** key **in** slownik:*
<instrukcje>

Pętla *for* również pozwala wykonać operacje na słownikach, zawracając jego klucz (*key*) .Gdy znamy klucz do słownika, to możemy wyświetlić wartość pod wskazanym kluczem. Przykład wykorzystania pętli *for* do pracy ze słownikami została pokazany na Rys. 26.

Rys. 26. Przykład użycia pętli *for* dla słowników.

Źródło: Opracowanie własne

Pętle można zagnieżdżać (czyli w pętli może umieszczać kolejne pętle) i łączyć z innymi instrukcjami, np. instrukcją warunkową *if*, jak zostało to pokazane Rys. 27.



Rys. 27. Przykład połączenia pętli *for* i instrukcji warunkowej *if*.

Źródło: Opracowanie własne

Powyższy przykład wyświetla elementy listy, jednak tylko te, które są większe od 5.

Pętla while

Pęta *for* wykonuje się z góry określona liczbę razy albo dla wszystkich elementów z listy, słownika, jednak jesteśmy wstanie określić, ile razy ta pętla się wykona. W programowaniu są jednak takie sytuacje, że nie wiem z góry, ile razy mają się wykonać instrukcje w pętli, wówczas możemy wykorzystać pętle *while*, która będzie się wykonywać tak długo, jak warunek w niej będzie prawdziwy. Pętla *while*, czyli *„dopóki”* tak samo jak instrukcja *if* sprawdza pewien warunek oraz ma podane instrukcje do wykonania. Różnica pomiędzy instrukcją *if* a *while* jest taka, że instrukcja *if* wykona operacje w niej zawarte jedne raz, gdy warunek jest prawdziwy, a pętla *while* tak długa jak warunek będzie prawdziwy. Czyli pętla *while* sprawdza warunek, wykonuje instrukcje, znowu sprawdza warunek, znowu wykonuje instrukcje i robi to tak długo, dopóki warunek jest prawdziwy. Jeśli warunek będzie fałszywy (nieprawdziwy) instrukcje nie zostaną wykonane ani razu, a program przejdzie do dalszej części programu. Szkielet pętli *while* wygląda następująco:

***while** (warunek):*
instrukcje 1
instrukcje 2

Warto znowu przypomnieć, że instrukcje, które mają być zawarte w pętli muszą być wcięte w stosunku do słowa kluczowego *while*. Przykład użycia pętli *while* jest pokazany na Rys. 28.

28.py

```
# Pętla while
a = 5

while a > 0:
    print (f"Wartość zmiennej a : {a}")
    a = a - 1
```

```
Wartość zmiennej a : 5
Wartość zmiennej a : 4
Wartość zmiennej a : 3
Wartość zmiennej a : 2
Wartość zmiennej a : 1
```

Rys. 28. Przykład wykorzystania pętli *while*.

Źródło: Opracowanie własne

Na początku zmienna *a* ma wartość 5, ona posłuży do sterowania pętlą, która będzie się wykonywała tak długo, jak wartość zmiennej będzie większa niż 0. Mamy zwarte dwie instrukcje w pętli: wyświetlamy wartość zmiennej i zmniejszamy wartość zmiennej o 1. Omawiając pętle *while* warto wspomnieć o dwóch instrukcjach: *continue* i *break*. Instrukcja *continue* – pomija wykonanie instrukcji i powoduje przejście do kolejnej iteracji (obrotu pętli), z kolei instrukcja *break* – powoduje przerwanie wykonywanie całej pętli. Powyższe instrukcje można stosować zarówno z pętlą *for*, jak i *while*. Rys. 29. pokazuje jak działa instrukcja *continue*, w tym przypadku na ekranie konsoli zostaną wyświetlone liczby 0 i 2 z listy *liczby*. Pozostałe liczby nie zostaną wyświetlone, gdyż operacja *continue* przerywa obieg pętli, gdy liczba *x* będzie mniejsza od 0.

29.py

```
# instrukcja continue
liczby = [-2, -1, 0, -4, 2]
for x in liczby:
    if x < 0:
        continue
    print(x)
```

Rys. 29. Przykład wykorzystania instrukcji *continue*.

Źródło: Opracowanie własne

Na Rys. 30. pokazany jest przykład użycia instrukcji *break*, która w tym konkretnym przypadku spowoduje, że w konsoli zostaną wyświetlone jedynie liczby -2 i -1, gdyż *break* przerywa działanie całej pętli, gdy *x* przyjmie wartość równą 0.

30.py

```
1  #instrukcja break
2  liczby = [-2, -1, 0, 1, 2]
3
4  for x in liczby:
5      if x == 0:
6          break
7      print(x)
```

Rys. 30. Przykład wykorzystania instrukcji *break*.

Źródło: Opracowanie własne

Funkcje w Pythonie

Do tej pory korzystaliśmy tylko z gotowych funkcji (tak na naprawdę metod, ale o tym później), których Python dostarcza olbrzymią ilość. W tym miejscu jednak nauczymy się tworzyć własne funkcje. Jest to bardzo proste i jednocześnie bardzo przyśpiesza tworzenie nowych programów, ponieważ raz napisany kod można bardzo łatwo i szybko wykorzystać ponownie. Nową funkcję deklarujemy używając słowa kluczowego **def** od podania jej nazwy oraz parametrów, jakie funkcja będzie pobierać, jeśli w ogóle jakieś ma pobierać:

def *<nazwa funkcji>():*
instrukcje 1
instrukcje 2

Jeżeli funkcja ma zwraca wartość, to wówczas będzie miała postać:

def *<nazwa funkcji>():*
instrukcje 1
instrukcje 2
return *wartość*

Poniżej mamy przykład funkcji *suma*, która dodaje dwie liczby. Argumentami funkcji są właśnie sumowane liczby (x, y). Funkcja nic nie zwraca, a jedynie wyświetla wynik sumowania liczb.

```
def suma(x, y):
    z = x + y
    print(z)
```

Tą samom funkcję można również napisać w taki sposób, by zamiast wyświetlania sumy liczb zwracała wynik.

```
def suma(x, y):
    z = x + y
    return z
```

Należy jednak pamiętać, że jeżeli funkcja zwraca wartość, to należy ją przypisać do jakieś zmiennej. Poniżej przykład wywołania funkcji zwracające wartość:

```
a = 3
b = 2
c = suma(a, b)
print(c)
```

Funkcja w tym konkretny przypadku zwraca wartość 5, gdyż sumuje dwie liczby zapisane w zmiennych *a* i *b* (3, 2). Do zmiennej *c* zostanie przypisana wartość zwracana przez funkcję, a funkcja *print* na ekranie wyświetli wartość 5. Funkcję, wywołujemy tak samo jak każdą inną, czyli używając jej nazwy, a w nawiasy wpisując parametry. Parametrami funkcji mogą być zarówno zmienne, jak i stałe. W większości przypadków funkcja zwraca jeden wynik, jednak należy pamiętać, że w Pythonie funkcja może zwrócić ich wiele jednocześnie. Zostało to pokazane na przykładzie poniżej.

```
def licz(x, y):
    z = x - y
    m = x ** y
    r = x + y
    return z, m, r
```

Poniżej wywołanie funkcji z argumentami:

```
wynik = licz(10, 5)
print(wynik[0], wynik[1], wynik[2])
```

Wywołanie funkcji wymaga podania wartości dla wszystkich parametrów - jeżeli nie podamy wartości dla wszystkich parametrów formalnych, wystąpi błąd. Możemy tego uniknąć, podając domyślne wartości argumentów- więcej o tym wątku zostanie powiedziane podczas szkolenia.

Wyjątki

Do tej pory zawsze zakładaliśmy, że nasz kod programu działa poprawnie (np. dzieląc liczby zakładaliśmy, że nikt nie będzie chciał dzielić przez zero). Co się jednak dzieje, kiedy coś pójdzie nie tak? W takich sytuacjach ”wyrzucany” jest wyjątek. Wyjątek jest obiektem specjalnego typu, który powoduje awaryjne przerwanie wykonania programu. Dla przykładu: wyjątek jest ”wyrzucany” np. kiedy staramy się odwołać do nieistniejącego elementu w liście, będziemy próbować dzielić przez zero itd. Jeżeli spróbujemy wykonać kod:

```
lista = [1, 2, 3, 4]
print (lista[5])
```

to wówczas pojawi się wyjątek, jak pokazano na Rys. 2.31.

```
C:\kurs_analiza\Scripts\python.exe C:\kurs_analiza\przy.py
Traceback (most recent call last):
  File "C:\kurs_analiza\przy.py", line 3, in <module>
    print (lista[5])
          ~~~~~^^^
IndexError: list index out of range
```

Rys. 31. Przykład wyjątku podczas odwołania się do nieistniejącego elementu w liście

Źródło: Opracowanie własne

Jak widać na Rys. 31. ostatnia linia informuje, jakiego rodzaju błąd wystąpił. „IndexError" jest typem wyjątku, który oznacza, że numer indeksu, do którego próbujemy się odwołać, jest niepoprawny. Z kolei po dwukropku następuje słowny opis błędu, który w tym przypadku informuje, że podaliśmy zbyt dużą cyfrę jako indeks listy. „Wyrzucony" wyjątek może zostać przez program złapany i obsłużony. Kiedy wyjątek jest „wyrzucany", wykonanie programu jest przerywane i wyjątek jest wyrzucany tak długo, aż zostanie obsłużony lub dopóki nie będzie już nic powyżej i wtedy program kończy swoje działanie z błędem. Wyjątki obsługuje się specjalną składnią, która wygląda następująco:

```
try:
        instrukcja1
        instrukcja2
        ...
except:
        instrukcja1
        instrukcja2
```

Dla naszego powyższego przykładu, obsługa wyjątku wygadałaby następująco:

```
try:
        a[3]
except:
        print('poza zakresem listy')
```

Taki kod zadziała, „wyrzucany" wyjątek zostanie obsłużony, jednak w ogólności lepiej zapisać obsługę wyjątku w bardziej ogólnej formie (niezależnie od wartości indeksu i z konkretnym typem wyjątku) :

```
lista = [1, 2, 3, 4]
indeks = 5
try:
        print(lista[indeks])
except IndexError:
        print(lista[len(lista)-1])
```

Konstrukcja „*try ... except ...”* może mieć na końcu dołożoną opcjonalną część *„else”*, która będzie wywoływana, jeśli kod wewnątrz sekcji „*try”* zostanie wykonany poprawnie bez wyrzucania wyjątku.

```
try:
        print(tablica[indeks])
except IndexError:
        print(tablica[len(tablica)-1])
else:
        print("Kod w bloku try został wykonany poprawnie")
```

Można jeszcze dołożyć jedną opcje (*finally*), która wykona się niezależnie, czy powstanie wyjątek, czy też nie.

```
try:
        print(tablica[indeks])
except IndexError:
        print(tablica[len(tablica)-1])
else:
        print("Kod w bloku try został wykonany poprawnie")
finally:
        print("Ten print wykona się zawsze bez względu na to czy
        powstanie wyjątek czy nie")
```

Elementy programowania obiektowego

Programowanie obiektowe różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów. W programowaniu obiektowym programista może deklarować własne typy zmiennych, tak zwane *klasy*, które mają w sobie *pola*, czyli *własności* oraz *zachowanie*, czyli *metody*. Na podstawie wzorca (szablonu), jakim jest *klasa* programista tworzy nowe *obiekty*.

Klasy definiujemy według następującego schematu:

```
class NaszaNowaKlasa:
    pola
    metody
```

Z kolei obiekty tworzymy według następującego schematu:

```
NazwaObiektu = NazwaKlasy(argumenty)
```

Ważnym elementem używania obiektów jest notacja obiektowa. Do pól i metod obiektów dostajemy się pisząc nazwę zmiennej dowiązanej do obiektu, kropkę i nazwę atrybutu obiektu.

```
nazwaObiektu.nazwaMetody()
nazwaObiektu.nazwaMetody(argumenty)
```

Kolejnym ważnym elementem klasy jest zmienna *self*.
Wewnątrz metod, zmienna *self* odnosi się do samego obiektu.

- Dzięki temu możliwy jest dostęp do pól obiektu, np. *self.a.*
- W momencie wywołania metody obiektu, zostaje on automatycznie wstawiony jako pierwszy argument metody i użytkownik podaje o jeden mniej argument niż metoda wymaga.

Przykład klasy przedstawiony jest poniżej:

```
class Wektor():
    def __init__(self, x, y):
        self.a = x
        self.b = y
        print "wektor został stworzony!"

w1 = Wektor(5, 7)      # wektor został stworzony
```

Tak jak zmienna *self* daje dostęp do pól obiektu, tak metoda *__init__* jest wywoływana automatycznie w momencie tworzenia obiektu. Metoda ta powinna wykonywać wszystkie operacje potrzebne do zainicjowania nowego obiektu, w szczególności powinna ona nadawać wartości jego polom. Oprócz specjalnej metody *__init__* programista może tworzyć własne metody. Metoda jest to funkcja zdefiniowana wewnątrz klasy.

```
class NazwaKlasy:
    def NazwaMetody(self, atrybuty):
        [ciało metody]
```

Wywołanie metody odbywa się następująco:

```
NazwaObiektu = NazwaKlasy(atr1, ..., atrN)      # tworzenie obiektu
NazwaObiektu.NazwaMetody(atrybuty)              # wywołanie metody na obiekcie
```

Przykładowe zadanie z rozwiązaniem

Napisz program, który posłuży do przechowania studentów w liście. Utwórz klasę Student:

- Pola: imię, nazwisko, oceny
- Metody: dodajOcene(ocena), wypiszOceny(), policzSrednia()

Utwórz menu: 1 - dodaj studenta, 2 - pokaż studentów, 3 - usuń studenta, 4 - dodaj ocenę studentowi, 5 - wypisz oceny studenta, 6 - średnia studenta, 7 - koniec.

Przykład klasy Student pokazany jest na Rys. 32., natomiast dalsza część programu z powyższego zadania przedstawiona jest na Rys. 33. i Rys. 34.

```python
class Student:
    def __init__(self, imie, nazwisko):
        self.imie=imie
        self.nazwisko=nazwisko
        self.oceny=[]

    1 usage
    def dodajOcene(self, ocena):
        self.oceny.append(ocena)

    1 usage
    def wypisz(self):
        for x in self.oceny:
            print(x)

    1 usage
    def policzSrednia(self):
        suma=0
        for x in self.oceny:
            suma=suma+x

        srednia=suma/len(self.oceny)

        print(srednia)
```

Rys. 32. Przykład klasy Student.

Źródło: Opracowanie własne

```
23  listaStudentow = []
24  while True:
25      menu = int(input("1-dodaj studenta, "
26                       "2-pokaz studentow, "
27                       "3-usun studenta, "
28                       "4-dodaj ocene studentowi, "
29                       "5-wypisz oceny studenta, "
30                       "6-srednia studenta, "
31                       "7-koniec"))
32
33      if menu == 1:
34          imie = input("Podaj imie: ")
35          nazwisko = input("Podaj nazwisko: ")
36          student=Student(imie,nazwisko)
37          listaStudentow.append(student)
38
39      elif menu == 2:
40          for x in listaStudentow:
41              print(f"imie: {x.imie}, Nazwisko: {x.nazwisko}")
42
43      elif menu == 3:
44          nazwisko = input("Podaj nazwisko: ")
45          for x in listaStudentow:
46              if x.nazwisko==nazwisko:
47                  listaStudentow.remove(x)
```

Rys. 33. Obsługa programu z klasą Student, cześć 1.

Źródło: Opracowanie własne

```
48
49      elif menu == 4:
50          nazwisko = input("Podaj nazwisko: ")
51          for x in listaStudentow:
52              if x.nazwisko == nazwisko:
53                  ocena = int(input("Podaj ocena: "))
54                  x.dodajOcene(ocena)
55
56      elif menu == 5:
57          nazwisko = input("Podaj nazwisko: ")
58          for x in listaStudentow:
59              if x.nazwisko == nazwisko:
60                  x.wypisz()
61
62      elif menu == 6:
63          nazwisko = input("Podaj nazwisko: ")
64          for x in listaStudentow:
65              if x.nazwisko == nazwisko:
66                  x.policzSrednia()
67
68      elif menu == 7:
69          break
70
```

Rys. 33. Obsługa programu z klasą Student, cześć 2.

Źródło: Opracowanie własne

Programowanie obiektowe, ma wiele aspektów, które zostaną bardziej szczegółowo omówione podczas szkolenia , min. hermetyzacja, dziedziczenie, i wiele innych.

Wprowadzenie do operacji na plikach

Zmienne stanowią pewien sposób przechowywania informacji i uzyskiwania do nich dostępu w trakcie wykonywania programu, jednak po wyłączeniu programu informacje ulatują. Dlatego warto zapisać dane w taki sposób, aby można je było później odzyskać. Do takiego

trwałego zapisu można wykorzystać pliki. Pliki można otworzyć na kilka sposobów (z różnymi atrybutami) w zależności od potrzeby.

- **Odczytywanie danych z plików tekstowych**

```
text_file = open("odczyt.txt", "r")
text_file.close()
```

Powyższe instrukcje pozwalają na otwarcie pliku o nazwie *odczyt.txt* z atrybutem ***r***, czyli do odczytu. Po tej instrukcji możemy czytać dane z pliku. Po zakończeniu czytania należy plik zamknąć. Plik możemy czytać na kilka sposobów, np. linia po linii. Poniżej został przedstawiony przykładowy kod programu, który czyta 3 linie pliku o nazwie *odczyt.txt*, a następnie wyświetla je na ekranie. W tym przykładzie czytamy plik po jednym wierszu naraz. *text_file* to taki uchwyt do pliku.

```
text_file = open("odczyt.txt", "r")
print(text_file.readline())
print(text_file.readline())
print(text_file.readline())
text_file.close()
```

Możemy również wczytać cały pliku do listy, a następnie wyświetlić je linia po linii. Metoda *readline()*, czyta jedną linie, z kolei *readlines()* czyta wszystkie linie z pliku.

```
text_file = open("odczyt.txt", "r")
lines = text_file.readlines()
print(lines)              # wszystkie linie od razu
print(len(lines))         # liczba linii
for line in lines:        # tak długo jak są linie
        print(line)       # linia po linii
text_file.close()
```

Tak naprawdę można to zrobić bez instrukcji *readlines()*. Przykład kodu jest zaprezentowany poniżej.

```
text_file = open("odczyt.txt", "r")
for line in text_file:
        print(line)
text_file.close()
```

Wybrane tryby dostępu do pliku tekstowego:

F = open("plik.txt","tryb")

- „*r*” - Odczyt danych z pliku tekstowego. Jeśli plik nie istnieje, zasygnalizuje błąd.
- „*w*” - Zapis danych do pliku tekstowego. Jeśli plik już istnieje, jego zawartość zostaje zastąpiona przez nowe dane. Jeśli nie istnieje, zostaje utworzony.
- „*a*” - Dopisanie danych na końcu pliku tekstowego. Jeśli plik istnieje, nowe dane zostają do niego dopisane. Jeśli plik nie istnieje, jest tworzony.

- **Zapisywanie łańcuchów znaków do pliku**

```
text_file = open("zapisz.txt", "w")
text_file.write("Wiersz 1\n")
text_file.write("To jest wiersz 2\n")
text_file.write("Ten tekst tworzy wiersz 3\n")
text_file.close()
```

Metody *write()* zapisuje łańcuch znaków do pliku. Warto wiedzieć, że metoda *write()* nie wstawia automatycznie znaku nowego wiersza na końcu łańcucha, który zapisuje. Należy samemu wstawić znaki nowego wiersza tam, gdzie są one potrzebne. Podobnie jak *readlines()*, metoda *writelines()* obsługuje listę łańcuchów, lecz zamiast wczytywać zawartość pliku tekstowego do listy, zapisuje listę łańcuchów do pliku.

```
text_file = open("zapisz_to.txt", "w")
lines = ["Wiersz 1\n",
"To jest wiersz 2\n",
"Ten tekst tworzy wiersz 3\n"]
text_file.writelines(lines)
text_file.close()
```

Wybrane metody do obsługi pliku

- *close()* - Zamyka plik. Odczytywanie danych z zamkniętego pliku oraz zapisywanie do niego jest niemożliwe, dopóki nie zostanie ponownie otwarty.
- *readline()* - Metoda zwraca wszystkie znaki od pozycji bieżącej do końca wiersza.
- *readlines()* - Odczytuje wszystkie wiersze pliku i zwraca je jako elementy listy.
- *write(dane)* - Zapisuje łańcuch dane do pliku.
- *writelines(dane)* - Zapisuje łańcuchy będące elementami listy dane do pliku.

Przykład programu z obsługą plików

Napisz program do wprowadzania studentów w zakresie informacji (imię, nazwisko, grupa). Program ma przechowywać dane w pliku txt, ma umożliwiać dodawanie, usuwanie, zmianę oraz pokazywanie listy studentów. Usuwanie oraz zmianę mogą wykonywać np. po nazwisku. Program wyposażony jest w interaktywne menu (1-dodaj, 2-pokaz, 3-usuń, 4-zmień, 5-wyjście).

Przykładowa realizacja zadania z obsługą plików jest pokazana na Rys. 34. i Rys. 35. W przypadku zapisu do pliku bardziej złożonych informacji, np. list, słowników, a nawet baz danych służy biblioteka *pickle*. Moduł *pickle* umożliwia marynowanie i przechowywanie w pliku bardziej złożonych danych. Przykład marynowania danych zostanie pokazany Państwu podczas zajęć.

przy.py

```
while True:
    menu = int(input("1-dodaj, 2-pokaz, 3-usun, 4-zmien, 5-koniec"))

    if menu == 1:
        imie = input("Podaj imie : ")
        nazwisko = input("Podaj nazwisko: ")
        grupa = input("Podaj grupa: ")
        plik=open("85.txt","a")
        plik.write(f"{imie};{nazwisko};{grupa}\n")
        plik.close()

    elif menu == 2:
        plik = open("85.txt", "r")
        for i in plik:
            iSplit=i.split(";")
            print(f"Imie:{iSplit[0]} Nazwisko:{iSplit[1]} Grupa:{iSplit[2]}")
        plik.close()

    elif menu == 3:
        listaTmp =[]
        plik = open("85.txt", "r")
        nazwisko = input("Podaj nazwisko: ")
        for i in plik:
            iSplit = i.split(";")
            if iSplit[1] != nazwisko:
                listaTmp.append(i)
        plik.close()
        plik = open("85.txt", "w")
        plik.writelines(listaTmp)
        plik.close()

```

Rys. 34. Program z obsługą plików, cześć 1.

Źródło: Opracowanie własne

```
32      elif menu == 4:
33          listaTmp = []
34          plik = open("85.txt", "r")
35          nazwisko = input("Podaj nazwisko: ")
36          for i in plik:
37              iSplit = i.split(";")
38              if iSplit[1] != nazwisko:
39                  listaTmp.append(i)
40              else:
41                  noweImie = input("Podaj nowe imie: ")
42                  noweNazwisko = input("Podaj nowe nazwisko: ")
43                  listaTmp.append(f"{noweImie};{noweNazwisko};{iSplit[2]}")
44          plik.close()
45          plik = open("85.txt", "w")
46          plik.writelines(listaTmp)
47          plik.close()
48
49      elif menu == 5:
50          break
```

Rys. 35. Program z obsługą plików, cześć 2.

Źródło: Opracowanie własne

Pliki binarne

Pliki binarne to pliki, w których dane zapisywane są w postaci zer i jedynek (0/1), a nie jako tekst czytelny dla człowieka (jak np. pliki .**txt** czy .**csv**). Człowiek nie jest w stanie „przeczytać" takiego pliku w edytorze tekstu, ale program komputerowy potrafi odtworzyć z niego obiekt w pamięci. W uczeniu maszynowym model po treningu jest obiektem w pamięci RAM, zawierającym: wyuczone parametry, współczynniki, strukturę modelu, informacje o preprocessingu. Po zamknięciu programu model znika, jeśli go nie zapiszemy. Korzystając z biblioteki **pickle** możemy w łatwy sposób zapisać obiekt (model) do pliku oraz go w późniejszym czasie załadować do programu. Zapis modelu do pliku za pomocą biblioteki **pickle** prezentuje Rys. 36. Otwieramy plik (**b** – binarny, **w** – do zapisu) i w nim zapisujemy za pomocą metody **dump** wytrenowany model. Z kolei Rys. 37. prezentuje w jaki sposób można odczytać modle z pliku binarnego za pomocą biblioteki **pickle**.

```python
import pickle

# zapis modelu do pliku binarnego
with open("model.pkl", "wb") as file:
    pickle.dump(model, file)
```

Rys. 36. Zapis modelu do pliku za pomocą biblioteki pickle.

Źródło: Opracowanie własne

```python
with open("model.pkl", "rb") as file:
    loaded_model = pickle.load(file)
```

Rys. 37. Odczyt modelu z pliku za pomocą biblioteki pickle.

Źródło: Opracowanie własne

Teraz za pomocą metody **load** z biblioteki **pickle** możemy otworzyć plik (**b** – binarny, **r** – do odczytu) i wyczytać model do obiektu o nazwie **loaded_model**, a następnie z niego korzystać.

Zadania (Python)

Zadanie 1

Utwórz przykładowy komentarz jednoliniowy i wielowierszowy (blokowy).

Zadanie 2

Utwórz zmienne o dowolnej nazwie, którym przypiszesz wartości: 80, 27.5, Kurs Python.

Zadanie 3

Napisz program, który wykona sumę cen produktów dla konkretnego zamówienia.

Cennik:

- chleb (5,40zł / 1 szt.),
- masło (6,50 zł / 1 szt.),
- pierniki (13,09 / 1kg.),
- sok (4,5 / 1 litr).

Zamówienie: 2 szt. chleba + 3 szt. masła + 1,5 kg pierników + 1 sok.

Zadanie 4

Samochód na 100 km spala 5,2 l paliwa. Ile spali paliwa po przejechaniu 479 km? Wykorzystaj zmienne i operatory w języku Python w celu obliczenia zadania.

Zadanie 5

Napisz program, który prosi użytkownika o podanie imienia i następnie wypisze na ekran powitanie po imieniu użytkownika, np. *„Witaj Paweł na programowaniu z Pythona”*.

Zadanie 6

Napisz program, który obliczy pole trójkąta na podstawie danych podanych przez użytkownika z konsoli tj.: wysokość (*h*) i długość podstawy tego trójkąta (*a*). Uwzględnij fakt, że wysokość i długość podstawy mogą być liczbami niecałkowitymi. Wzór na obliczeni pola

$$P_{\Delta} = \frac{1}{2}ah.$$

Zadanie 7

Napisz program obliczający średnią z pięciu liczb podanych przez użytkownika. Liczby mogą być typu zmiennoprzecinkowego.

Zadanie 8

Napisz interaktywny sklep z trzema produktami:
Chleb – 6.50 zł
Sok – 4.00 zł
Pączek – 5.50 zł
Użytkownik będzie pytany o ilość dla każdej z ww. pozycji asortymentowej, ilość musi być całkowita (*int*). Wypisz podsumowanie zakupów, czyli co zostało kupione, ile sztuk i jaka wartość. Wypisz, ile należy zapłacić całkowicie za złożone zamówienie.

Zadanie 9

Napisz program do nauki tabliczki mnożenia. Program ma wylosować dwie liczby z zakresu (1-10), po czym ma zapytać użytkownika, jaki będzie wynik mnożenia tych liczb. Użytkownik

podaje swój wynik. Natomiast po podaniu wyniku przez użytkownika, program wyświetla swój wynik. Póki co nie sprawdzamy, czy wynik podany przez użytkownika jest poprawny.
Np.

*Ile to jest 3 * 7 ?*
Odpowiedz użytkownika: 21
Odpowiedź komputera: 21

Zadanie 10

Napisz program, który będzie obliczał potęgę. Potęga zostanie obliczona na podstawie pobranych danych od użytkownika tj. podstawa i wykładnika ($podstawa^{wykladnik}$).

Zadanie 11

Zaprojektuj program, który wczyta od użytkownika dowolny tekst. Program za zadanie wypisać:

- Ile prowadzono znaków,
- Ile jest spacji w wprowadzonym tekście.

Np.: „Programowanie w Pythonie”
Liczba znaków: 24
Liczba spacji: 2

Zadanie 12

Napisz program, który 5 razy poprosi o podanie imienia. Podane imiona będą zapisywane do listy. Wypisz dla wszystkich imion z listy poniższy komunikat:
Cześć <tutaj imię z listy>

Zadanie 13

Napisz program, w którym zadeklarujesz dwie listy, które będą przechowywały po 4 dowolne liczby. Np.:

lista1 = [1, 3, 2, 5]
lista2 = [4, 5, 1, 8]

Program powinien wyświetlić sumę wszystkich liczb z obu list (29).

Zadanie 14

Napisz program, w którym użytkownik podaje 7 dowolnych liczb całkowitych i dodaje je do listy. Program ma za zadanie:

- wyświetlić wszystkie liczby,
- policzyć sumę wszystkich liczb
- policzyć średnią
- odwrócić kolejność elementów w liście.

Zadanie 15

Napisz program, który 5 razy poprosi użytkownika o wprowadzenie dowolnych liczb całkowitych. Program za zadanie zliczyć, ile wprowadzono liczb unikatowych. Pomocne mogą okazać się zbiory.

Zadanie 16

Wykorzystując poznane typy sekwencyjne zaprojektuj kod dla poniższej funkcjonalności:
Utwórz zmienne z wartościami:

- zmienna1 = ”jeden”
- zmienna2 = ”pięć”
- zmienna3= ”siedem”

Oblicz sumę ww. zmiennych. Suma zmiennych, to: 13. W rozwiązaniu problemu pomocne mogą okazać się słowniki.

Zadanie 17

Zaprojektuj program, która dowolną liczbę 4-cyfrową zamieni na
interpretację słowną np.:

- 9112 # dziewięć jeden jeden dwa
- 5842 # pięć osiem cztery dwa

W rozwiązaniu problemu pomocne mogą okazać się słowniki.

Zadanie 18

Napisz program do nauki tabliczki mnożenia. Program ma wylosować dwie liczby z zakresu (1-10), po czym ma zapytać użytkownika, jaki będzie wynik mnożenia tych liczb. Użytkownik podaje swój wynik. Natomiast po podaniu wyniku przez użytkownika, program wyświetla swój wynik. Program ma również sprawdzić, czy wynik podany przez użytkownika jest poprawny. Np.

*Ile to jest 3 * 7 ?*
Odpowiedź użytkownika: 23
Odpowiedź komputera: 21
Użytkowniku poddałeś błędny wynik!

Zadanie 19

Napisz program, który sprawdza, czy wprowadzona liczba jest liczbą parzystą, czy nieparzystą. Wykorzystaj instrukcję modulo, czyli resztę z dzielenia **%**.

Zadanie 20

Utwórz 2 zmienne, przypisując im dowolne - różne wartości liczbowe. Napisz program, który wskaże największą wartość. Rozszerz program dodając dodatkową zmienną (trzecią) i przypisz jej dowolną wartość różną od powyższych, następnie wskaże największą wartość.

Zadanie 21

Utwórz 3 zmienne i przypisz im dowolne wartości liczbowe np.: $a = 10$ $b = 2$ $c = 9$. Wypisz wartości zmiennych od największej do najmniejszej w konsoli.

Zadanie 22

Napisz program, który oblicza wartość współczynnika BMI wg wzoru (*waga* / *wzrost***2). Wzrost podawany jest w metrach. Jeżeli wynik jest w przedziale (18.5 – 24.9) to wypisuje w konsoli *„waga prawidłowa”*, jeżeli poniżej to *„niedowaga”*, jeżeli powyżej to *„nadwaga”*.

Zadanie 23

Napisz program, który spośród liczb 1-100 wyświetli tylko te, które są podzielne przez 6.

Zadanie 24

Napisz program, który pobiera od użytkownika np. 10 liczb i oblicza sumę tylko tych liczb, które są nieparzyste.

Zadanie 25

Utwórz listę i dodaj do niej w pętli 8 imion. Następnie, wypisz imiona z listy zgodnie z poniższym wzorem: Witaj <imię z listy>.

Zadanie 26

Napisz własny mechanizm obliczania potęgi (bez użycia operatora potęgowania **). Użytkownik podaje podstawę i wykładnik potęgi. W celu napisania powyższego mechanizmu wykorzystaj pętle. Warto jeszcze przypomnieć, że wszystko co jest podniesione do potęgi 0 jest równe 1, a wykładnikiem potęgi są liczby większe bądź równe 0.

Zadanie 27

Napisz program, który oblicza silnie. Użytkownik podaje dowolną liczbę całkowitą dodatnią, a program zwraca wartość silni z podanej liczby. Np. 5! = 1 * 2 * 3 * 4 * 5 = 120.

Zadanie 28

Napisz grę: komputer losuje liczbę z przedziału 1 – 100. Użytkownik ma za zadanie odgadnąć, co to za liczba poprzez podawanie kolejnych wartości. Jeżeli podana liczba jest: większa od wylosowanej - wyświetlany jest komunikat „podałeś za dużą liczbę”, mniejsza od wylosowanej – wyświetlony jest komunikat „podałeś za małą liczbę”, Równą wylosowanej – wyświetlony jest komunikat „Gratulacje” i gra zostaje zakończona.

Rozwiązania zadań

Zadanie 1

```
# przykład użycia instrukcji print
print("Witaj Świecie!")

"""
Programowanie
w
Pytonie
jest
fajne"""
print("Do pracy ...")
```

Zadanie 2

```
wiek = 80
cenaCukierkow = 27.5
kurs_programowania = "Kurs Python."
```

Zadanie 3

```
cenaChleba = 5.40
cenaMasla = 6.50
cenaPierniki = 13.09
cenaSok = 4.5

zamowienie = 2*cenaChleba + 3*cenaMasla + 1.5*cenaPierniki + 1*cenaSok
print(f"Zamówienie: 2 szt. chleba + 3 szt. masła + 1,5 kg pierników + 1 sok
ma wartość: {zamowienie} zł")
```

Zadanie 4

```
ile_na_100 = 5.2
droga = 479
spalanie = (droga/100)*ile_na_100
print(f"Po przejechaniu {droga} km samochód spali {spalanie} litrów
paliwa.")
```

Zadanie 5

```
imie = input("Podaj imie: ")
print(f"Witaj {imie} na programowaniu z Pythona.")
```

Zadanie 6

```
print("Program oblicza pole trójkąta!")
h = float(input("Podaj wyskość trójkąta: "))
a = float(input("Podaj długość podstawy trójkąta: "))

pole = 0.5*a*h
print(f"Pole trójkąta o wykości {h} i długości podstawy {a} wynosi: {pole}
")
```

Zadanie 7

```
print("Program oblicza średnią z pięciu zadeklarownych liczb.")
l1 = 1.0
l2 = 2.0
l3 = 3.0
l4 = 4.0
l5 = 5.0
suma = l1 + l2 + l3 + l4 + l5
srednia = suma/5.0
print(f"Średnia z liczb: {l1}, {l2}, {l3}, {l4}, {l5}, to: {srednia}")
```

Zadanie 8

```
print("Sklep")
chleb = 6.50
sok = 4.00
paczek = 5.50
ile_chlebow = int(input("Ile sztuk chleba chcesz zamówić? "))
ile_sokow = int(input("Ile sztuk soków chcesz zamówić? "))
ile_paczkow = int(input("Ile pączków chcesz zamówić? "))
wartosc_chleb = ile_chlebow * chleb
wartosc_sok = ile_sokow * sok
wartosc_paczkow = ile_paczkow * paczek
print("")
print(f"Zamówiłeś {ile_chlebow} chlebów o wartości: {wartosc_chleb}")
print(f"Zamówiłeś {ile_sokow} soków o wartości: {wartosc_sok}")
print(f"Zamówiłeś {ile_paczkow} pączków o wartości: {wartosc_paczkow}")
zamownienie = wartosc_chleb + wartosc_sok + wartosc_paczkow
print(f"Całkowita wartość zamówienia, to: {zamownienie}")
```

Zadanie 9

```
import random #biblioteka do liczb pseudolosowych

l1=random.randint(1,10) #losowanie liczby całkowitej z zakresu <1,10>
l2=random.randint(1,10)
wynik=l1*l2
liczba_u=input(f"Ile to jest {l1} * {l2} ?")

print(f"Odpowiedź użytkownika: {liczba_u}")
print(f"Odpowiedź komputera: {wynik}")
```

Zadanie 10

```
print("Potęgowanie liczb.")
pod=float(input("Podaj podstawę: "))
wyk=float(input("Podaj wykładnik: "))

wy=pod**wyk
print(f"Wynik wynosi: {wy}.")
```

Zadanie 11

```
tekst = input("Wprowadź tekst: ")
ile_znakow = len(tekst)          #len() sprawdza długość napisu
ile_spacji = tekst.count(" ")    #tekst.count(" ") zlicza liczbę wystąpień
spacji w napisie tekst
print(f"Liczba znaków: {ile_znakow}")
print(f"Liczba spacji: {ile_spacji}")
```

Zadanie 12

```
lista=[] #przykład bez pętli
print("Dodawanie do listy imion!")
lista.append(input("Podaj 1 imie: "))
lista.append(input("Podaj 2 imie: "))
lista.append(input("Podaj 3 imie: "))
lista.append(input("Podaj 4 imie: "))
lista.append(input("Podaj 5 imie:"))
print("Lista imion: ")
print(lista)
```

```
print("")
print(f"Cześć {lista[0]}")
print(f"Cześć {lista[1]}")
print(f"Cześć {lista[2]}")
print(f"Cześć {lista[3]}")
print(f"Cześć {lista[4]}")
```

Zadanie 13

```
#Program sumuje wszystkie liczby z obu list, wersja bez pętli.
lista1 = [1, 3, 2, 5]
lista2 = [4, 5, 1, 8]
suma =
lista1[0]+lista1[1]+lista1[2]+lista1[3]+lista2[0]+lista2[1]+lista2[2]+lista
2[3]
print(f"Suma liczb to {suma}")
```

Zadanie 14

```
#Wersja programu z pętlą
lista=[]
suma = 0
for i in range(1,8):
    lista.append(int(input(f"Podaj {i} liczbę: ")))
    suma = suma + i

print()
print("Elementy listy: ")
print(lista)

print()
srednia=suma/len(lista)
print(f"Suma liczb to {suma}")
print(f"Średnia liczb to {srednia}")

print()
print("Elementy listy w odwrotnej kolejności: ")
for i in range(6, -1, -1):
    print(lista[i], end=", ")
```

Zadanie 15

```
#Zbiory z założenia przechowują unikatowe dane
#Wersja bez pętli
zbior = set()
zbior.add(int(input("Podaj liczbę: ")))
zbior.add(int(input("Podaj liczbę: ")))
zbior.add(int(input("Podaj liczbę: ")))
zbior.add(int(input("Podaj liczbę: ")))
zbior.add(int(input("Podaj liczbę: ")))
ile = len(zbior) #ilość elementów zbioru = liczba unikatowych liczb

print(f"Unikatowych liczb jest {ile}")
```

Zadanie 16

```
zmienna1 = "jeden"
zmienna2 = "pięć"
zmienna3 = "siedem"
slownik={"jeden":1, "pięć":5, "siedem":7}

suma=slownik[zmienna1] + slownik[zmienna2] + slownik[zmienna3]
print(f"Suma liczb: {suma} ")
```

Zadanie 17

```
slownik = {"1":"jeden", "2":"dwa", "3":"trzy", "4":"cztery", "5":"pięć",
"6":"sześć", "7":"siedem", "8":"osiem", "9":"dziewięć"}
liczba = input("Podaj liczbe 4 cyforową :")

print(f"{slownik[liczba[0]]} {slownik[liczba[1]]} {slownik[liczba[2]]}
{slownik[liczba[3]]}")
```

Zadanie 18

```
import random # biblioteka do generowania liczb pseudolosowych

l1=random.randint(1,10) # losowanie liczby z zakresu <1, 10>
l2=random.randint(1,10)
wynik=l1*l2
liczba_g=int(input(f"Ile to jest {l1} * {l2} ?"))
```

```
print(f"Odpowiedź użytkownika: {liczba_g}")
print(f"Odpowiedź komputera: {wynik}")

if wynik == liczba_g:
    print("Użytkowniku podałeś prawidłowy wynik!")
else:
    print("Użytkowniku podałeś błędny wynik!")
```

Zadanie 19

```
liczba = int(input("Wprowadź liczbę: "))
if liczba % 2 == 0:
    print("Wprowadzona liczba jest parzysta!")
else:
    print("Wprowadzona liczba jest nieparzysta!")
```

Zadanie 20

```
# wersja 2 zmienne
zm1 = 5.0
zm2 = 2.0
print (f"Wartość zm1 = {zm1}, a zm2 = {zm2}")
if zm1 > zm2:
    print(f"Zmienna pierwsza ma większą wartość: {zm1}!")
else:
    print(f"Zmienna druga ma większą wartość: {zm2}!")

# wersja 3 zmienne (założenie zmienne są różne!)
z1 = 2.0
z2 = 3.0
z3 = 10.0
print (f"Wartość z1 = {z1}, z2 = {z2}, z3 = {z3}")
if z1 > z2:
    if z1 > z3:
        print("Zmienna z1 największa!")
    else:
        print("Zmienna z3 największa!")
elif z2 > z3:
    print("Zmienna z2 jest największa!")
else:
    print("Zmienna z3 jest największa!")
```

Zadanie 21

```
#wersja bez wbudwanych funkcji
a = 10
b = 2
c = 9

if a > b and a >c:
    print(a, end=" ")
    if b > c:
        print(b, c)
    else:
        print(c, b)
elif b > c and b > c:
    print(b, end=" ")
    if a > c:
        print(a, c)
    else:
        print(c, a)
elif c > a and c > b:
    print(c, end=" ")
    if a > b:
        print(a, b)
    else:
        print(b, a)
```

Zadanie 22

```
waga = float(input("Podaj swoją wagę [kg]: "))
wzrost = float(input("Podaj swój wzrost [m]: "))

bmi = waga/(wzrost**2)
print(f"Twoje BMI: {bmi}")
print()

if bmi >= 18.5 and bmi <=24.9:
    print("Waga prawidłowa!")
elif bmi <18.5:
    print("Niedowaga!")
```

```
else:
    print("Nadwaga!")
```

Zadanie 23

```
print("Program wyświetla liczby podzielne przez 6 z zakresu od <1,100>. ")
for i in range (1,101):
    if i%6 == 0:
       print(i, end=", ")
```

Zadanie 24

```
suma = 0
for i in range(1, 11):
    liczba = int(input(f"Podaj {i} liczbę: "))
    if liczba % 2 == 1:
       suma = suma + liczba

print(f"Suma liczb nieparzystych, to: {suma}")
```

Zadanie 25

```
imiona = []
for i in range (8):
    imiona.append(input("Podaj imie: "))

print()

for i in imiona:
    print(f"Witaj {i}.")
```

Zadanie 26

```
podstawa = int(input("Podaj podstawę: "))
wykladnik = int(input("Podaj wykładnik: "))

potega = 1

if wykladnik >= 0:
    for i in range(wykladnik):
        potega=potega*podstawa
    print(f"{podstawa}^{wykladnik}={potega}")
```

```
else:
    print("Nieprawidłowa wartość wykładnika!")
```

Zadanie 27

```
liczba = int(input("Podaj z jakiej liczby chcesz obliczyć silnie: "))
silnia = 1
if liczba >= 0:
    for i in range(1,liczba+1):
        silnia=silnia*i
    print(f"{liczba}! = {silnia}")
else:
    print("Wyznaczenie silni dotyczy liczb dodatnich oraz zera.")
```

Zadanie 28

```
import random
los = random.randint(1,100)

while True:
    liczba = int(input("Podaj wylosowaną liczbę: "))

    if liczba > los:
        print("Podałeś za dużą liczbę.")
    elif liczba < los:
        print("Podałeś za małą liczbę.")
    elif liczba == los:
        print("Gratulacje!")
        break
```

Analiza danych i uczenie maszynowe

Co to jest analiza danych i uczenie maszynowe?

Analiza danych to proces przekształcania surowych danych w informację, wiedzę i wnioski, które mogą wspierać podejmowanie decyzji. W praktyce nie polega ona wyłącznie na obliczeniach, ale na zrozumieniu danych, ich jakości, struktury oraz zależności, jakie między nimi występują. W rzeczywistych projektach dane rzadko są idealne. Zazwyczaj są: niekompletne, zawierają błędy, pochodzą z różnych źródeł, zapisane w różnych formatach.

Dlatego analiza danych obejmuje kilka kluczowych etapów:

1. Pozyskiwanie danych (pliki CSV, Excel, bazy danych, API),
2. Czyszczenie danych (braki, duplikaty, błędy),
3. Eksplorację danych (szukanie zależności i wzorców),
4. Wizualizację danych,
5. Formułowanie wniosków i rekomendacji.

Przykłady analizy danych w praktyce:

- analiza sprzedaży w firmie (które produkty sprzedają się najlepiej),
- analiza danych klientów (kto odchodzi, kto kupuje częściej),
- analiza wyników egzaminów lub ankiet,
- analiza ruchu na stronie internetowej,
- analiza danych finansowych.

Uczenie maszynowe (Machine Learning) jest naturalnym i logicznym rozszerzeniem analizy danych, w którym celem nie jest już jedynie opisanie i zrozumienie danych historycznych, lecz budowa modeli zdolnych do generalizacji wiedzy i podejmowania decyzji na podstawie nowych, wcześniej niewidzianych danych. Modele uczenia maszynowego uczą się zależności występujących w danych na podstawie przykładów, a następnie wykorzystują zdobytą wiedzę do przewidywania przyszłych wartości, klasyfikowania obiektów, identyfikowania wzorców oraz wykrywania nietypowych obserwacji.

W przeciwieństwie do klasycznego programowania, w którym logika działania systemu jest definiowana ręcznie w postaci sztywnych reguł i instrukcji, w uczeniu maszynowym reguły te powstają automatycznie w procesie uczenia. Oznacza to, że zamiast opisywać krok po kroku, jak program ma podejmować decyzje, dostarczamy mu dane wejściowe oraz oczekiwane rezultaty, a algorytm samodzielnie wyznacza relacje pomiędzy zmiennymi. Takie podejście pozwala tworzyć rozwiązania, które są elastyczne, odporne na zmienność danych i zdolne do adaptacji w dynamicznych środowiskach.

Model uczenia maszynowego stanowi matematyczny opis zależności występujących w danych, zapisany w postaci parametrów, wag lub struktur decyzyjnych. Jakość tego modelu zależy nie tylko od zastosowanego algorytmu, lecz w dużej mierze od jakości danych, sposobu

ich przygotowania oraz poprawnej oceny wyników. W praktyce oznacza to, że skuteczne uczenie maszynowe wymaga połączenia wiedzy z zakresu programowania, statystyki oraz analizy danych, a sam proces budowy modelu jest iteracyjny i wymaga testowania różnych podejść.

Istotnym aspektem uczenia maszynowego jest również umiejętność oceny, na ile wyuczony model radzi sobie z nowymi danymi. Modele, które zbyt dokładnie dopasowują się do danych treningowych, mogą tracić zdolność generalizacji, natomiast zbyt proste modele nie są w stanie uchwycić istotnych zależności. Dlatego w praktycznych zastosowaniach uczenia maszynowego równie ważne jak samo trenowanie modelu jest jego walidowanie, interpretacja wyników oraz świadome wykorzystanie w rzeczywistych systemach analitycznych i decyzyjnych. Dlatego proces uczenia maszynowego obejmuje kilka kluczowych etapów:

1. Zdefiniowanie problemu i celu modelu (co chcemy przewidywać lub klasyfikować),
2. Przygotowanie danych do modelowania (selekcja cech, kodowanie zmiennych, skalowanie),
3. Podział danych na zbiory treningowe i testowe,
4. Dobór i trenowanie modelu uczenia maszynowego,
5. Ocena jakości modelu oraz jego optymalizacja,
6. Wykorzystanie modelu do predykcji i podejmowania decyzji.

Przykłady zastosowań uczenia maszynowego w praktyce:

- przewidywanie cen nieruchomości lub produktów,
- prognozowanie sprzedaży i popytu,
- klasyfikacja klientów (np. lojalny / zagrożony odejściem),
- wykrywanie nadużyć i anomalii w danych,
- systemy rekomendacyjne i personalizacja treści.

Podział technik uczenia maszynowego

Można wyróżnić wiele technik uczenia maszynowego oraz uczenia głębokiego, które różnią się zakresem zastosowań, stopniem złożoności oraz wymaganiami obliczeniowymi. Ze względu na ograniczenia czasowe szkolenia, w ramach niniejszego kursu omówione zostaną jedynie wybrane techniki, najczęściej wykorzystywane w praktycznych projektach analizy danych i uczenia maszynowego. Podczas zajęć uczestnicy krok po kroku poznają pełny proces

pracy z danymi – od wczytania i przygotowania danych, przez ich analizę i wizualizację w postaci licznych wykresów, aż po budowę, ocenę i interpretację wyników modeli uczenia maszynowego. Szczególny nacisk zostanie położony na zrozumienie całego przepływu pracy, tak aby prezentowane rozwiązania mogły być łatwo przenoszone na inne zbiory danych, modyfikowane poprzez zmianę modelu lub jego parametrów oraz dostosowywane do własnych potrzeb i problemów analitycznych. Poniżej dokonamy podziału technik uczenia maszynowego.

1. **Uczenie nadzorowane (Supervised Learning)**

Uczenie nadzorowane jest najczęściej stosowaną i najbardziej intuicyjną techniką uczenia maszynowego, szczególnie w projektach biznesowych i analitycznych. Jego istotą jest uczenie modelu na danych, które zawierają zarówno zmienne wejściowe, jak i znaną poprawną odpowiedź, nazywaną zmienną docelową lub etykietą. Model otrzymuje więc przykłady typu: „takie dane → taki wynik" i na ich podstawie uczy się zależności, które następnie może wykorzystać do przewidywania wyników dla nowych danych.

Proces uczenia polega na stopniowym dopasowywaniu parametrów modelu w taki sposób, aby minimalizować błąd pomiędzy przewidywaniami modelu a rzeczywistymi wartościami. Kluczową cechą uczenia nadzorowanego jest możliwość obiektywnej oceny jakości modelu, ponieważ dla danych testowych znamy poprawne odpowiedzi i możemy sprawdzić, jak dobrze model generalizuje wiedzę.

Uczenie nadzorowane dzieli się na dwa główne typy problemów: regresję oraz klasyfikację. W regresji model przewiduje wartości liczbowe, natomiast w klasyfikacji przypisuje obiekty do określonych klas. Technika ta jest szczególnie dobrze dopasowana do sytuacji, w których dysponujemy dużą liczbą historycznych danych oraz jasno zdefiniowanym celem predykcji.

Przykłady problemów rozwiązywanych za pomocą uczenia nadzorowanego:

- przewidywanie cen mieszkań na podstawie ich cech,
- prognozowanie sprzedaży w kolejnych miesiącach,
- klasyfikacja klientów jako lojalnych lub zagrożonych odejściem,
- wykrywanie spamu w wiadomościach e-mail,
- ocena ryzyka kredytowego klienta.

2. Uczenie nienadzorowane (Unsupervised Learning)

Uczenie nienadzorowane jest techniką, w której model pracuje na danych pozbawionych etykiet, czyli takich, dla których nie znamy poprawnych odpowiedzi. Celem nie jest tutaj przewidywanie konkretnej wartości, lecz odkrywanie struktury, wzorców i zależności ukrytych w danych. Model samodzielnie analizuje dane i próbuje znaleźć w nich naturalne grupy, podobieństwa lub nietypowe obserwacje.

Technika ta jest szczególnie użyteczna na etapie eksploracji danych, gdy nie mamy jeszcze jasno określonego celu predykcyjnego lub gdy chcemy lepiej zrozumieć charakter analizowanego zbioru danych. Uczenie nienadzorowane często stanowi punkt wyjścia do dalszych analiz lub do budowy modeli nadzorowanych, np. poprzez segmentację danych. Jednym z największych wyzwań uczenia nienadzorowanego jest interpretacja wyników, ponieważ brak etykiet uniemożliwia klasyczną ocenę jakości modelu. Wymaga to większego zaangażowania analityka oraz dobrej znajomości kontekstu biznesowego lub problemowego.

Przykłady problemów rozwiązywanych za pomocą uczenia nienadzorowanego:

- segmentacja klientów na podstawie ich zachowań zakupowych,
- grupowanie produktów o podobnych cechach,
- wykrywanie anomalii i nietypowych transakcji,
- analiza zachowań użytkowników na stronie internetowej,
- redukcja wymiarowości i wizualizacja złożonych danych.

3. Uczenie przez wzmacnianie (Reinforcement Learning)

Uczenie przez wzmacnianie jest techniką, która znacząco różni się od pozostałych podejść. W tym przypadku model, nazywany agentem, uczy się poprzez interakcję ze środowiskiem, podejmując kolejne decyzje i obserwując ich konsekwencje. Zamiast zbioru danych wejściowych z etykietami, agent otrzymuje sygnał zwrotny w postaci nagrody lub kary, na podstawie którego modyfikuje swoje zachowanie. Celem uczenia przez wzmacnianie jest znalezienie takiej strategii działania, która maksymalizuje sumę nagród w długim okresie. Proces uczenia jest iteracyjny i często wymaga wielu prób oraz symulacji, co sprawia, że technika ta jest bardziej złożona obliczeniowo i koncepcyjnie niż uczenie nadzorowane i nienadzorowane.

Uczenie przez wzmacnianie znajduje zastosowanie głównie tam, gdzie decyzje są sekwencyjne, a ich skutki ujawniają się w czasie. Choć rzadziej spotykane w klasycznych kursach analizy danych, stanowi ważny element nowoczesnej sztucznej inteligencji.

Przykłady problemów rozwiązywanych za pomocą uczenia przez wzmacnianie:

- systemy sterowania i robotyka,
- gry komputerowe i planszowe,
- optymalizacja tras i harmonogramów,
- systemy rekomendacyjne uczące się na bieżących interakcjach,
- zarządzanie zasobami i procesami w czasie rzeczywistym.

4. Uczenie półnadzorowane (Semi-supervised Learning)

Uczenie półnadzorowane łączy elementy uczenia nadzorowanego i nienadzorowanego. W tym podejściu tylko część danych posiada etykiety, a pozostała część jest nieoznaczona. Technika ta jest szczególnie przydatna w sytuacjach, gdy pozyskanie etykiet jest kosztowne, czasochłonne lub wymaga wiedzy eksperckiej. Model wykorzystuje niewielką liczbę oznaczonych przykładów, aby nauczyć się podstawowych zależności, a następnie wspomaga się dużą ilością danych nieoznaczonych w celu poprawy jakości predykcji. Takie podejście często pozwala uzyskać lepsze wyniki niż klasyczne uczenie nadzorowane przy ograniczonej liczbie etykiet.

Przykłady problemów rozwiązywanych za pomocą uczenia półnadzorowanego:

- analiza danych medycznych,
- klasyfikacja dokumentów i tekstów,
- rozpoznawanie obrazów,
- systemy rekomendacyjne oparte na ograniczonej liczbie ocen,
- przetwarzanie danych pochodzących z Internetu.

Każda z technik uczenia maszynowego odpowiada innemu typowi problemów i innemu charakterowi danych. W praktyce analitycznej kluczowe jest nie tylko poznanie algorytmów, lecz przede wszystkim umiejętność doboru właściwej techniki do konkretnego problemu. W ramach kursu uczestnicy poznają te podejścia w praktyce, ucząc się, jak świadomie wykorzystywać je w analizie danych i projektach uczenia maszynowego.

Biblioteki i narzędzia w Pythonie

Praktyczna analiza danych i uczenie maszynowe w Pythonie opierają się na wykorzystaniu sprawdzonych narzędzi oraz bibliotek, które wspierają każdy etap pracy z danymi – od ich wczytania, przez przygotowanie i analizę, aż po budowę oraz ocenę modeli uczenia maszynowego. Współczesne projekty analityczne rzadko polegają na pisaniu rozwiązań od podstaw. Zamiast tego korzysta się z rozbudowanego ekosystemu bibliotek, które znacząco przyspieszają pracę, zwiększają czytelność kodu oraz pozwalają skupić się na rozwiązywaniu problemów, a nie na implementacji niskopoziomowych mechanizmów.

W ramach niniejszego kursu głównym środowiskiem pracy będzie **Google Colab**, a podstawowymi formatami danych będą pliki **CSV** oraz **Excel (XLSX)**. Takie podejście odzwierciedla realia pracy analityka danych oraz specjalisty uczenia maszynowego, gdzie dane bardzo często pochodzą z plików eksportowanych z systemów biznesowych, baz danych lub narzędzi raportowych.

- ✓ **Środowisko pracy: Google Colab**

Google Colab to środowisko analityczne działające w przeglądarce internetowej, oparte na technologii Jupyter Notebook. Umożliwia ono pisanie i uruchamianie kodu Pythona bez konieczności instalowania oprogramowania na komputerze lokalnym. Jest to szczególnie istotne w kontekście szkoleń praktycznych, ponieważ eliminuje problemy związane z konfiguracją środowiska oraz różnicami systemowymi pomiędzy uczestnikami.

Colab oferuje gotowe środowisko z zainstalowanym Pythonem oraz najważniejszymi bibliotekami do analizy danych i uczenia maszynowego. Notebooki pozwalają na wykonywanie kodu krok po kroku, obserwowanie wyników pośrednich, tworzenie wykresów oraz dokumentowanie analizy za pomocą komentarzy tekstowych. Taki sposób pracy sprzyja nauce, eksperymentowaniu oraz lepszemu zrozumieniu przetwarzanych danych i działania modeli.

W praktyce Google Colab pełni rolę interaktywnego laboratorium, w którym możliwe jest szybkie testowanie różnych rozwiązań, modyfikowanie kodu oraz natychmiastowa obserwacja efektów w postaci tabel, wykresów i wyników predykcji.

✓ **Format danych: CSV i Excel**

Podstawowym źródłem danych w kursie będą pliki w formatach **CSV** oraz **XLSX**. Są to jedne z najczęściej spotykanych formatów danych w analizie danych, wykorzystywane zarówno w środowiskach biznesowych, jak i naukowych. Pliki CSV charakteryzują się prostą strukturą i dużą uniwersalnością, natomiast pliki Excel są powszechnie używane do raportowania, zestawień oraz ręcznej obróbki danych.

W kontekście analizy danych kluczowe jest umiejętne wczytywanie tych plików do środowiska Python oraz ich dalsze przetwarzanie. Dane zapisane w plikach często zawierają braki, niespójności, różne typy danych lub nadmiarowe informacje, które wymagają oczyszczenia i odpowiedniego przygotowania przed dalszą analizą i modelowaniem.

✓ **Pandas – fundament pracy z danymi**

Podstawową biblioteką wykorzystywaną do pracy z danymi tabelarycznymi jest **Pandas**. Umożliwia ona wczytywanie danych z plików CSV i Excel, ich przechowywanie w strukturach danych takich jak **DataFrame** oraz wykonywanie operacji analitycznych i transformacji danych.

Pandas pozwala m.in. na:

- selekcję i filtrowanie danych,
- obsługę brakujących wartości,
- sortowanie i grupowanie danych,
- łączenie danych z różnych źródeł,
- tworzenie nowych cech na podstawie istniejących danych.

W praktycznych projektach analizy danych to właśnie Pandas zajmuje największą część pracy analityka. Odpowiednie przygotowanie danych w tej bibliotece ma bezpośredni wpływ na jakość dalszych analiz oraz skuteczność modeli uczenia maszynowego.

✓ **NumPy – obliczenia numeryczne**

NumPy jest biblioteką wspierającą obliczenia numeryczne oraz operacje na wielowymiarowych tablicach danych. Choć często działa „w tle” i nie jest bezpośrednio widoczna dla użytkownika, stanowi fundament wielu innych bibliotek analitycznych i uczenia maszynowego.

W kontekście kursu NumPy będzie wykorzystywana do operacji matematycznych, pracy z macierzami oraz efektywnego przetwarzania danych liczbowych. Jej znaczenie rośnie wraz z wielkością zbiorów danych oraz złożonością obliczeń.

- ✓ **Wizualizacja danych: Matplotlib i Seaborn**

Wizualizacja danych jest nieodłącznym elementem analizy danych oraz uczenia maszynowego. Pozwala ona lepiej zrozumieć strukturę danych, rozkłady zmiennych, zależności pomiędzy cechami oraz jakość działania modeli.

Do tworzenia wykresów wykorzystywane będą biblioteki **Matplotlib** oraz **Seaborn**. Matplotlib zapewnia pełną kontrolę nad wyglądem wykresów, natomiast Seaborn upraszcza tworzenie estetycznych i czytelnych wizualizacji statystycznych. W trakcie kursu wykresy będą wykorzystywane zarówno na etapie eksploracji danych, jak i do interpretacji wyników modeli uczenia maszynowego.

- ✓ **Scikit-learn – uczenie maszynowe w praktyce**

Kluczową biblioteką kursu w obszarze uczenia maszynowego jest **scikit-learn**. Jest to biblioteka zaprojektowana z myślą o praktycznym zastosowaniu algorytmów uczenia maszynowego, oferująca spójny i intuicyjny interfejs do trenowania, testowania oraz oceny modeli. Scikit-learn umożliwia:

- budowę modeli regresji i klasyfikacji,
- realizację algorytmów uczenia nienadzorowanego,
- przygotowanie danych do modelowania (skalowanie, kodowanie),
- podział danych na zbiory treningowe i testowe,
- ocenę jakości modeli za pomocą różnych metryk,
- tworzenie pipeline'ów łączących przetwarzanie danych i modelowanie.

Biblioteka ta doskonale nadaje się do nauki uczenia maszynowego, ponieważ pozwala skupić się na logice problemu i interpretacji wyników, a nie na implementacji algorytmów od podstaw. Jednocześnie rozwiązania oparte na scikit-learn są powszechnie stosowane w projektach komercyjnych i analitycznych.

Podczas kursu uczestnicy będą stopniowo przechodzić przez pełny proces analizy danych i uczenia maszynowego, wykorzystując opisane narzędzia w sposób spójny

i praktyczny. Praca będzie odbywać się na rzeczywistych zbiorach danych, a każdy etap – od wczytania plików CSV i Excel, przez analizę i wizualizację, aż po budowę i ocenę modeli – będzie realizowany w Google Colab z użyciem bibliotek Pythona. Prezentowane rozwiązania będą miały charakter uniwersalny, tak aby mogły być łatwo przenoszone na inne dane, modyfikowane poprzez zmianę modelu lub parametrów oraz dostosowywane do własnych potrzeb analitycznych.

Ogólny algorytm analizy danych i uczenia maszynowego

Krok 1. Przygotowanie środowiska i instalacja bibliotek

Pierwszym etapem jest przygotowanie środowiska pracy tak, aby każdy uczestnik miał możliwość uruchomienia tego samego kodu i uzyskania porównywalnych wyników. W kursach praktycznych bardzo wygodnym rozwiązaniem jest Google Colab, ponieważ eliminuje problemy instalacyjne i pozwala od razu przejść do pracy z danymi. Należy upewnić się, że podstawowe biblioteki są dostępne, a w razie potrzeby doinstalować je w notebooku. Warto od razu ustalić standard pracy: jedna komórka na importy, jedna na ustawienia (np. losowość), a następnie logiczne sekcje notebooka. Dobrą praktyką jest też zapisywanie wersji bibliotek (np. w komentarzu), aby w przyszłości łatwiej było odtworzyć wyniki. Jeśli projekt ma charakter zespołowy, warto już na starcie zadbać o czytelne nazewnictwo plików, folderów i notebooków.

Krok 2. Zdefiniowanie celu analizy i problemu ML

Zanim wczytasz dane, musisz jasno określić, co jest celem projektu. W analizie danych celem może być znalezienie zależności, odpowiedź na pytanie biznesowe lub wyciągnięcie wniosków z danych historycznych. W uczeniu maszynowym celem jest zwykle predykcja (regresja) albo klasyfikacja (np. 0/1), ewentualnie grupowanie (clustering) lub wykrywanie anomalii. Na tym etapie warto nazwać zmienną docelową, opisać co oznacza „dobry wynik" oraz jakie błędy są kosztowne (np. fałszywie negatywne w medycynie). Ten krok wpływa na dobór metryk, algorytmu oraz sposobu przygotowania danych. Dobrze też wskazać, jakiego typu dane spodziewasz się zobaczyć (liczbowe, kategoryczne, daty) i jakie mogą występować ograniczenia (np. brak etykiet, mała liczba obserwacji).

Krok 3. Pozyskanie danych i wczytanie do Pythona (CSV/XLSX)

Kolejnym etapem jest pozyskanie danych i wczytanie ich do środowiska analitycznego, najczęściej do obiektu DataFrame w Pandas. W praktyce kursowej bardzo często pracuje się na plikach CSV i Excel, więc kluczowe jest poprawne ustawienie separatorów, kodowania oraz interpretacji typów danych. Już na tym etapie trzeba zwrócić uwagę, czy nagłówki kolumn są poprawne, czy nie ma „pustych" kolumn, oraz czy dane nie zostały źle wczytane jako tekst zamiast liczb. Warto też sprawdzić rozmiar zbioru danych, liczbę kolumn oraz pierwsze rekordy, aby upewnić się, że struktura jest zgodna z oczekiwaniami. Dobrą praktyką jest natychmiastowe zapisanie krótkiej informacji o źródle danych oraz krótkiego opisu, co zawiera zbiór.

Krok 4. Szybki przegląd danych (sanity check)

Zanim rozpoczniesz jakiekolwiek przekształcenia, wykonuje się tzw. sanity check, czyli podstawową kontrolę jakości danych. Obejmuje to sprawdzenie typów danych, braków danych, duplikatów, nietypowych wartości oraz podstawowych statystyk opisowych. Ten krok bardzo często pozwala wykryć problemy na wczesnym etapie, np. kolumny z wartościami „0", które w rzeczywistości oznaczają brak pomiaru, albo błędne jednostki. W praktyce, im szybciej wykryjesz problem z danymi, tym mniej czasu stracisz później na „naprawianie" modelu, który tak naprawdę uczy się na błędnych danych. To także etap, na którym warto zidentyfikować potencjalną zmienną docelową oraz wstępnie ocenić, czy klasy są zbalansowane (w klasyfikacji).

Krok 5. Czyszczenie danych i przygotowanie jakościowe

Czyszczenie danych to etap, w którym usuwamy lub korygujemy elementy, które mogłyby zaburzyć analizę i modelowanie. Typowe działania to uzupełnianie lub usuwanie braków danych, usuwanie duplikatów, poprawa typów danych (np. zamiana tekstu na liczby), standaryzacja nazw kategorii oraz obsługa wartości odstających. Bardzo ważne jest, aby decyzje o czyszczeniu były uzasadnione, a nie przypadkowe, ponieważ każda taka decyzja wpływa na wynik modelu. W projektach szkoleniowych warto dopisywać komentarze: dlaczego dana kolumna jest usuwana, dlaczego braki są uzupełniane medianą albo czemu wartości „0" zamieniono na NaN. Ten krok jest często najdłuższy w całym procesie, ale ma największy wpływ na jakość końcowego rozwiązania.

Krok 6. Eksploracyjna analiza danych (EDA)

EDA (ang. *Exploratory Data Analysis*) to etap, w którym staramy się zrozumieć dane: jak wyglądają rozkłady zmiennych, jakie są zależności między cechami, oraz co różni poszczególne grupy obserwacji. W praktyce wykonuje się zarówno analizę liczbową (statystyki, korelacje), jak i wizualną (histogramy, boxploty, wykresy zależności). W EDA często wykrywa się problemy, które wcześniej nie były widoczne, np. silną skośność rozkładu, nielogiczne wartości albo kolumny silnie skorelowane. Dla projektów ML EDA jest też momentem, w którym można wstępnie ocenić, które cechy prawdopodobnie będą informatywne, a które mogą wnosić szum. Na końcu EDA warto sformułować kilka wniosków, które będą prowadzić do decyzji o preprocessingu i modelowaniu.

Krok 7. Przygotowanie danych do modelowania (feature engineering + preprocessing)

Gdy dane są już zrozumiane i oczyszczone, przechodzi się do przygotowania ich w formie „zjadliwej" dla algorytmu. Obejmuje to wybór cech, kodowanie zmiennych kategorycznych (np. one-hot encoding), skalowanie cech liczbowych oraz ewentualne transformacje (np. logarytmowanie). W tym kroku można też tworzyć nowe cechy, np. różnice, ilorazy, agregaty czy flagi logiczne, jeśli ma to sens w kontekście problemu. Ważne jest, aby preprocessing był wykonywany w sposób powtarzalny, najlepiej poprzez pipeline, aby podczas predykcji na nowych danych wykonać dokładnie te same kroki. W praktyce ML jest to fundament poprawnego wdrożenia modelu – bez spójnego preprocessingu model często „działa tylko w notebooku".

Krok 8. Podział danych na zbiory treningowe i testowe

Zanim zaczniemy trenować model, dane należy podzielić na zbiór treningowy i testowy. Zbiór treningowy służy do uczenia modelu, a testowy do sprawdzenia, jak model radzi sobie na danych, których nie widział wcześniej. Taki podział jest niezbędny do oceny generalizacji i ograniczenia ryzyka „oszukania się", że model działa dobrze, gdy w rzeczywistości jedynie zapamiętał dane treningowe. W zależności od problemu stosuje się też walidację krzyżową, szczególnie gdy zbiór danych jest mały. W klasyfikacji często dba się o zachowanie proporcji klas w podziale (stratyfikacja), aby wyniki były wiarygodne.

Krok 9. Wybór modelu bazowego i trening pierwszej wersji

Następnie buduje się pierwszy, prosty model bazowy, który stanowi punkt odniesienia. W praktyce często zaczyna się od modeli interpretowalnych, takich jak regresja logistyczna lub proste drzewa, aby zrozumieć problem i uzyskać pierwsze wyniki. Na tym etapie kluczowe jest poprawne dopasowanie modelu do rodzaju problemu (regresja vs klasyfikacja) i użycie właściwych danych wejściowych. Pierwszy model nie musi być najlepszy – jego celem jest sprawdzenie, czy pipeline działa oraz uzyskanie wstępnej oceny trudności problemu. To również moment, w którym ujawniają się typowe problemy, np. niezbalansowane klasy lub zbyt duża liczba cech w stosunku do liczby obserwacji.

Krok 10. Ewaluacja modelu i dobór metryk

Ocena modelu powinna być dopasowana do problemu. Dla klasyfikacji często liczy się nie tylko accuracy, ale również precision, recall, F1-score oraz macierz pomyłek, bo różne błędy mogą mieć różny koszt. Dla regresji stosuje się m.in. MAE, MSE, RMSE lub R^2, a interpretacja zależy od jednostek i skali danych. Ważne jest, aby na podstawie metryk odpowiedzieć na pytanie: czy model jest wystarczająco dobry dla celu projektu i co dokładnie oznacza „dobry". W projektach szkoleniowych warto uczyć kursantów interpretacji: np. wysoki accuracy przy niezbalansowanych klasach może być złudny, bo model może „zgadywać" klasę dominującą. Ewaluacja to również etap, w którym analizuje się błędy: na jakich przypadkach model się myli i dlaczego.

Krok 11. Poprawa modelu: tuning, inne algorytmy, balans klas

Po uzyskaniu wyników modelu bazowego przechodzi się do ulepszania rozwiązania. Może to oznaczać dobór lepszego algorytmu, dostrojenie hiperparametrów, zmianę preprocessingu, usunięcie nieistotnych cech lub dodanie nowych. W klasyfikacji częstym problemem jest niezbalansowanie klas, więc stosuje się metody takie jak wagi klas, undersampling/oversampling lub techniki pokroju SMOTE. W tym kroku kluczowe jest zachowanie metodologii: każdą zmianę należy ocenić w porównywalny sposób na danych testowych lub w walidacji, aby nie dopasować się przypadkowo do jednego podziału danych. To etap, który uczy najbardziej praktycznego myślenia: poprawa modelu to zwykle seria kontrolowanych eksperymentów.

Krok 12. Interpretacja wyników i wnioski

Dobry projekt ML nie kończy się na liczbie w metryce. Trzeba jeszcze zinterpretować, co wynik oznacza i czy model jest użyteczny w kontekście problemu. W praktyce analizuje się również znaczenie cech, wpływ preprocessingu oraz stabilność wyników. Jeśli model ma wspierać decyzje, warto jasno opisać, jakie decyzje mogą zostać podjęte na podstawie jego predykcji i jakie ryzyka się z tym wiążą. To też moment na wskazanie ograniczeń: np. mała liczba danych, brak istotnych zmiennych, możliwe błędy pomiarowe. Wnioski powinny być sformułowane tak, aby osoba nietechniczna mogła zrozumieć, co model robi i jak dobrze działa.

Krok 13. Zapis modelu i przygotowanie do użycia na nowych danych

W praktyce model powinien być zapisany do pliku (najczęściej binarnego), aby nie trenować go od nowa i móc wykorzystać go na nowych danych. Dobrą praktyką jest zapisywanie nie tylko samego modelu, ale całego pipeline'u (preprocessing + model), aby zapewnić spójność działania w przyszłości. Na tym etapie warto też przygotować przykładową funkcję „predict", która pokazuje, jak wczytać model i wykonać predykcję dla nowego rekordu danych. To jest bardzo ważny element dydaktyczny, bo pokazuje, że ML to nie tylko trening, ale również praktyczne wykorzystanie modelu.

W przedstawionych trzynastu krokach zaprezentowana została kompletna procedura tworzenia modelu uczenia maszynowego – od pracy z surowymi danymi, poprzez analizę i przygotowanie danych, aż do budowy, oceny oraz wykorzystania modelu. Należy podkreślić, że każdy problem analityczny jest inny i w praktyce sposób postępowania może się różnić w zależności od charakteru danych, celu analizy oraz dostępnych zasobów. Niemniej jednak, niezależnie od specyfiki projektu, proces analizy danych i uczenia maszynowego zazwyczaj przebiega w przybliżeniu według zaprezentowanego schematu, który stanowi uniwersalny i sprawdzony punkt odniesienia w pracy analityka danych oraz specjalisty uczenia maszynowego.

Dobór technik uczenia maszynowego do rodzaju problemu i danych

Jednym z kluczowych etapów pracy z uczeniem maszynowym jest świadomy dobór techniki modelowania, który wynika nie z mody na konkretny algorytm, lecz z charakteru problemu, struktury danych oraz celu analizy. W praktyce analityk danych bardzo rzadko

zaczyna pracę od pytania „jaki algorytm jest najlepszy", a znacznie częściej od pytania „jakie są moje dane i czego tak naprawdę od nich oczekuję".

W niniejszym rozdziale przedstawiono najważniejsze techniki uczenia maszynowego w sposób opisowy i kontekstowy, tak aby uczestnik szkolenia mógł zrozumieć nie tylko jak dany algorytm działa, ale przede wszystkim, kiedy i dlaczego warto go zastosować.

- ✓ **Regresja liniowa**

Regresja liniowa jest jednym z najstarszych i najprostszych modeli statystycznych, a jednocześnie stanowi punkt odniesienia dla niemal wszystkich problemów regresyjnych. Jej istota polega na modelowaniu zależności pomiędzy zmienną docelową, a zestawem cech wejściowych w postaci liniowej kombinacji tych cech. Model próbuje znaleźć takie współczynniki, aby możliwie najlepiej opisać trend występujący w danych.

W praktyce regresja liniowa bardzo rzadko jest modelem końcowym, ale pełni niezwykle istotną rolę poznawczą. Pozwala szybko sprawdzić, czy w danych istnieje jakakolwiek zależność pomiędzy cechami a zmienną docelową, a także umożliwia interpretację wpływu poszczególnych cech na wynik. Dzięki temu analityk może zrozumieć kierunek i siłę zależności, zanim przejdzie do bardziej złożonych technik.

Regresja liniowa najlepiej sprawdza się w sytuacjach, gdy dane są względnie czyste, relacje pomiędzy zmiennymi są w przybliżeniu liniowe, a celem projektu jest nie tylko predykcja, lecz również wyjaśnienie zjawiska. W projektach biznesowych bywa często wykorzystywana jako model wyjaśniający, nawet jeśli jego skuteczność predykcyjna jest niższa niż w przypadku modeli zespołowych.

- ✓ **Ridge, Lasso i ElasticNet**

W rzeczywistych zbiorach danych bardzo często spotyka się sytuację, w której liczba cech jest duża, a pomiędzy nimi występują silne zależności. W takich przypadkach klasyczna regresja liniowa staje się niestabilna, a wyuczone współczynniki mogą mieć nieintuicyjne wartości. Właśnie w tym miejscu pojawiają się techniki regularyzacji, takie jak Ridge, Lasso oraz ElasticNet. Modele te zachowują ideę regresji liniowej, ale wprowadzają dodatkowe ograniczenie, które „karze" zbyt duże wartości współczynników. W praktyce oznacza to, że model staje się bardziej odporny na szum, lepiej generalizuje oraz jest mniej podatny na

przeuczenie. Szczególnie istotna jest regresja Lasso, która ma zdolność do zerowania współczynników, co prowadzi do automatycznej selekcji cech.

Techniki regularyzacji są często stosowane w problemach, gdzie dane są wysokowymiarowe, liczba obserwacji jest ograniczona, a interpretowalność nadal ma znaczenie. Stanowią one naturalny krok po regresji liniowej w projektach, które wymagają większej stabilności modelu.

- ✓ **Regresja logistyczna**

Regresja logistyczna, mimo swojej nazwy, jest jedną z najważniejszych technik klasyfikacji, szczególnie binarnej. Jej celem nie jest bezpośrednie przypisanie klasy, lecz oszacowanie prawdopodobieństwa przynależności do danej klasy. Dzięki temu model ten bardzo dobrze wpisuje się w projekty decyzyjne, w których ważne jest nie tylko „co", ale również „z jaką pewnością". W praktyce regresja logistyczna jest niezwykle popularna w projektach biznesowych, finansowych i medycznych, ponieważ zapewnia kompromis pomiędzy skutecznością a interpretowalnością. Współczynniki modelu można analizować podobnie jak w regresji liniowej, co pozwala zrozumieć, które cechy zwiększają lub zmniejszają prawdopodobieństwo wystąpienia danego zdarzenia.

Regresja logistyczna bardzo często pełni rolę modelu referencyjnego, do którego porównuje się bardziej złożone algorytmy. Nawet jeśli zostaje później zastąpiona przez Random Forest, czy boosting, jej wyniki stanowią ważny punkt odniesienia.

- ✓ **Naive Bayes**

Modele Naive Bayes opierają się na probabilistycznym podejściu do klasyfikacji i zakładają niezależność cech względem siebie. Choć założenie to rzadko jest spełnione w rzeczywistych danych, w praktyce modele te potrafią działać zaskakująco dobrze, szczególnie w analizie tekstu i danych wysokowymiarowych. Technika ta jest często pierwszym wyborem w problemach takich jak klasyfikacja dokumentów, filtrowanie spamu czy analiza opinii. Jego siłą jest szybkość działania oraz zdolność do pracy na bardzo dużej liczbie cech, gdzie inne algorytmy miałyby trudności obliczeniowe.

W projektach szkoleniowych Naive Bayes jest doskonałym przykładem algorytmu, który pokazuje, że prosty model oparty na silnych założeniach może być bardzo skuteczny, jeśli jest dobrze dopasowany do problemu.

- ✓ **k-Nearest Neighbors**

Algorytm k-najbliższych sąsiadów opiera się na intuicyjnej idei podobieństwa obserwacji. Zamiast budować jawny model, algorytm przechowuje dane treningowe i podejmuje decyzję na podstawie tego, jakie etykiety mają najbardziej podobne obserwacje. Technika k-NN bardzo dobrze pokazuje, jak ważne w uczeniu maszynowym jest przygotowanie danych, w szczególności skalowanie cech. Bez odpowiedniego preprocessingu algorytm ten traci sens, ponieważ odległości pomiędzy obserwacjami przestają być miarodajne.

W praktyce k-NN stosowany jest głównie do problemów o niewielkiej lub średniej skali, często w celach edukacyjnych lub eksploracyjnych, gdzie jego prostota i intuicyjność są dużą zaletą.

- ✓ **Support Vector Machines**

SVM to technika, która koncentruje się na wyznaczeniu granicy decyzyjnej o maksymalnym marginesie pomiędzy klasami. W swojej istocie jest to algorytm geometryczny, który bardzo dobrze radzi sobie w przestrzeniach wysokowymiarowych. Dzięki zastosowaniu funkcji jądra SVM potrafi modelować złożone, nieliniowe zależności bez jawnego zwiększania liczby cech. W praktyce algorytm ten bywa stosowany w problemach, gdzie dane są trudne do separacji, a jakość klasyfikacji jest kluczowa.

Jednocześnie SVM wymaga dużej uwagi przy doborze parametrów i jest mniej skalowalny niż modele zespołowe, co sprawia, że w bardzo dużych projektach bywa zastępowany innymi technikami.

- ✓ **Drzewa decyzyjne**

Drzewa decyzyjne budują model w postaci sekwencji logicznych reguł. Każdy węzeł drzewa odpowiada decyzji opartej na wartości jednej z cech, a ścieżka od korzenia do liścia reprezentuje pełny proces decyzyjny. Ich największą wartością jest interpretowalność – model można przedstawić w formie logicznych reguł zrozumiałych nawet dla osób nietechnicznych. Z tego powodu drzewa decyzyjne są często wykorzystywane w projektach, gdzie transparentność modelu jest równie ważna jak jego skuteczność. W praktyce pojedyncze drzewo rzadko jest modelem końcowym, ponieważ łatwo ulega przeuczeniu. Stanowi jednak fundament dla potężniejszych technik zespołowych.

- ✓ **Random Forest**

Random Forest jest jedną z najczęściej stosowanych technik uczenia maszynowego w pracy z danymi tablicowymi. Łączy wiele drzew decyzyjnych w jeden model, którego predykcja jest wynikiem agregacji decyzji poszczególnych drzew. W praktyce Random Forest jest często pierwszym „poważnym" modelem, po który sięga analityk. Działa dobrze bez skomplikowanego preprocessingu, radzi sobie z nieliniowościami, a jednocześnie jest stosunkowo odporny na szum i overfitting.

- ✓ **Extra Trees – losowość jako sposób na generalizację**

Extra Trees są rozwinięciem idei Random Forest, w którym proces budowy drzew jest jeszcze bardziej losowy. Dzięki temu model uczy się bardziej zróżnicowanych reguł, co często prowadzi do lepszej generalizacji kosztem interpretowalności. W praktyce Extra Trees są szczególnie użyteczne w problemach z dużą liczbą cech oraz w sytuacjach, gdy dane są zaszumione.

- ✓ **Boosting**

Boosting to rodzina technik, w których modele budowane są sekwencyjnie, a każdy kolejny skupia się na poprawianiu błędów poprzedniego. Algorytmy takie jak Gradient Boosting, XGBoost czy LightGBM należą do najskuteczniejszych technik w pracy z danymi tablicowymi. W praktyce boosting jest często wybierany w projektach, gdzie maksymalna skuteczność predykcji ma kluczowe znaczenie, a złożoność modelu jest akceptowalna. Są to algorytmy wymagające doświadczenia, ale oferujące bardzo dużą kontrolę nad procesem uczenia.

Dobór techniki uczenia maszynowego nie jest decyzją jednorazową, lecz procesem iteracyjnym, w którym analityk stopniowo dopasowuje model do danych i celu projektu. Rozdział ten ma stanowić mapę myślenia, do której uczestnik kursu będzie wracał przy każdym nowym problemie analitycznym. Pewnego rodzaju podsumowaniem powyższego opisu jest Rys. 38., który przedstawia wybór techniki uczenia maszynowego w zależności od problemu i skuteczności techniki. Z kolei rysunek Rys. 39. obrazuje ogólny podział technik uczenia maszynowego ze względu na kategorię, Rys. 40. dokonuje podziału ML ze względu na problem i dane, do których ma zostać wykorzystany.

Rys. 38. Wybór technik uczenia maszynowego

Źródło: Opracowanie własne

Kategoria	Techniki
Regresja	Regresja liniowa, Ridge, Lasso, ElasticNet
Klasyfikacja liniowa	Regresja logistyczna, Linear SVM
Klasyfikacja nieliniowa	SVM (RBF), k-NN
Modele drzewiaste	Drzewa decyzyjne
Modele zespołowe (bagging)	Random Forest, Extra Trees
Modele zespołowe (boosting)	AdaBoost, Gradient Boosting, XGBoost, LightGBM
Metody probabilistyczne	Naive Bayes
Modele oparte na sąsiedztwie	k-NN
Redukcja wymiarowości	PCA
Klasteryzacja	k-means, DBSCAN, Hierarchical Clustering
Anomalie	Isolation Forest, One-Class SVM

Rys. 39. Ogólny podział technik uczenia maszynowego

Źródło: Opracowanie własne

Problem	Dane	Rekomendowane techniki
Regresja, małe dane	liniowe	Linear / Ridge / Lasso
Regresja, nieliniowa	średnie/duże	Random Forest, XGBoost
Klasyfikacja binarna	interpretacja	Logistic Regression
Klasyfikacja złożona	nieliniowa	RF, Extra Trees
Maks. skuteczność	tablicowe	XGBoost, LightGBM
Tekst	sparse	Naive Bayes, SVM
Segmentacja	brak etykiet	k-means, DBSCAN
Anomalie	rzadkie zdarzenia	Isolation Forest

Rys. 40. Podział technik uczenia maszynowego ze względu na problem (dane).

Źródło: Opracowanie własne

Należy pamiętać, że nie istnieje jeden najlepszy algorytm. Istnieje algorytm najlepiej dopasowany do danych i celu. W praktyce zaczynamy od prostych modeli, następnie analizujemy błędy, zwiększamy złożoność i porównujemy wyniki.

Pozyskiwanie danych do analizy danych i uczenia maszynowego

Każdy projekt analizy danych i uczenia maszynowego rozpoczyna się od danych. Bez danych nie ma ani analizy, ani modeli, ani wniosków. W praktyce bardzo często to pozyskanie odpowiednich danych stanowi największe wyzwanie, a nie sam wybór algorytmu, czy implementacja modelu. Dane mogą pochodzić z wielu źródeł, mieć różną jakość, format i poziom kompletności, a ich charakter w dużej mierze determinuje dalsze etapy pracy analitycznej. Warto podkreślić, że w rzeczywistych projektach bardzo rzadko spotyka się „gotowe" zbiory danych, które można bezpośrednio wykorzystać w modelu. Dlatego już na etapie nauki należy oswajać się z danymi surowymi. Dane do analizy i uczenia maszynowego mogą pochodzić zarówno z otwartych repozytoriów, jak i z systemów biznesowych, baz

danych, API czy plików raportowych. Bardzo dobrym źródłem danych na start jest portal Kaggle: https://www.kaggle.com/

- ✓ **Portal Kaggle**

Kaggle jest jednym z najpopularniejszych źródeł danych w społeczności data science i uczenia maszynowego. Platforma ta udostępnia ogromną liczbę zbiorów danych z różnych dziedzin, takich jak finanse, medycyna, marketing, sport, edukacja czy nauki społeczne. Dane dostępne na Kaggle są często dobrze opisane, zawierają dokumentację oraz przykładowe notebooki, co czyni je idealnym materiałem do nauki. W praktyce Kaggle jest doskonałym miejscem do:

- nauki pracy z danymi tablicowymi (CSV, Excel),
- ćwiczenia technik czyszczenia danych,
- testowania różnych modeli ML na tym samym problemie,
- budowania projektów do portfolio.

Warto jednak pamiętać, że dane z Kaggle bywają częściowo „przygotowane", dlatego nie zawsze w pełni oddają problemy spotykane w projektach komercyjnych.

- ✓ **Repozytoria danych akademickich i publicznych**

Wiele instytucji naukowych oraz organizacji publicznych udostępnia dane w ramach otwartych inicjatyw. Zbiory te często mają wysoki poziom wiarygodności i są wykorzystywane w badaniach naukowych.

Do najczęściej spotykanych źródeł należą:

- dane statystyczne publikowane przez urzędy państwowe,
- zbiory danych demograficznych i ekonomicznych,
- dane środowiskowe, klimatyczne i geograficzne,
- dane edukacyjne i społeczne.

Tego typu dane doskonale nadają się do projektów analizy danych, raportowania oraz budowy modeli predykcyjnych opartych na danych rzeczywistych. Ich dodatkową zaletą jest możliwość łączenia wielu źródeł w jeden zbiór, co pozwala ćwiczyć bardziej zaawansowane scenariusze analityczne.

✓ **Dane z plików: CSV i Excel**

Jednym z najczęstszych źródeł danych w praktyce analitycznej są pliki w formatach CSV oraz Excel. Dane takie pochodzą zazwyczaj z systemów biznesowych, narzędzi raportowych, eksportów z baz danych lub ręcznie przygotowanych zestawień.

Praca z plikami CSV i Excel jest szczególnie istotna, ponieważ:

- są to formaty powszechnie stosowane w firmach,
- dane często zawierają błędy, braki i niespójności,
- wymagają interpretacji struktury i znaczenia kolumn.

W projektach szkoleniowych pliki te idealnie nadają się do nauki wczytywania danych, eksploracji, czyszczenia oraz przygotowania do modelowania. Uczestnicy uczą się również, że sama obecność danych w pliku nie oznacza jeszcze, że dane są gotowe do analizy.

✓ **Dane z baz danych**

W rzeczywistych projektach bardzo często dane nie są przechowywane w plikach, lecz w bazach danych. Mogą to być relacyjne bazy danych (np. systemy transakcyjne) lub hurtownie danych. Choć na kursach podstawowych nie zawsze pracuje się bezpośrednio z bazami, warto uświadamiać uczestników, że w praktyce analityk bardzo często pobiera dane za pomocą zapytań.

Dane z baz danych:

- są zwykle duże i znormalizowane,
- wymagają agregacji i łączenia wielu tabel,
- często zawierają dane historyczne.

Umiejętność pracy z danymi pochodzącymi z baz jest niezwykle cenna i stanowi naturalne rozszerzenie kompetencji analitycznych.

✓ **Dane pobierane przez API**

Coraz więcej danych udostępnianych jest za pośrednictwem interfejsów API. Pozwala to na automatyczne pobieranie aktualnych danych, np. pogodowych, finansowych, społecznościowych czy logistycznych. Dane z API:

- często są w formacie JSON,
- wymagają przetwarzania do postaci tabelarycznej,
- pozwalają tworzyć dynamiczne projekty ML.

W kontekście uczenia maszynowego API umożliwiają pracę z danymi zmieniającymi się w czasie oraz symulowanie scenariuszy zbliżonych do produkcyjnych.

- ✓ **Dane generowane sztucznie (syntetyczne)**

W projektach edukacyjnych i testowych często wykorzystuje się dane syntetyczne, generowane programowo. Pozwalają one:

- kontrolować strukturę danych,
- symulować określone zależności,
- testować zachowanie algorytmów.

Dane syntetyczne są szczególnie przydatne do nauki działania algorytmów oraz do demonstracji problemów takich jak overfitting, niezbalansowane klasy czy wpływ szumu na model.

- ✓ **Dane własne i dane z rzeczywistych projektów**

Najlepszym materiałem do nauki są często dane własne lub dane pochodzące z realnych problemów. Mogą to być:

- dane sprzedażowe,
- dane ankietowe,
- dane pomiarowe,
- dane operacyjne.

Praca z takimi danymi uczy nie tylko technik analizy i ML, ale również rozumienia kontekstu biznesowego oraz interpretacji wyników w praktyce.

Pozyskiwanie danych jest pierwszym i jednym z najważniejszych etapów analizy danych oraz uczenia maszynowego. Źródło danych, ich jakość oraz sposób pozyskania mają bezpośredni wpływ na dalsze etapy pracy – od analizy eksploracyjnej, przez przygotowanie danych, aż po skuteczność modeli. W ramach kursu uczestnicy będą pracować głównie na danych pochodzących z plików CSV i Excel oraz z publicznych repozytoriów, ucząc się jednocześnie, jak samodzielnie wyszukiwać i oceniać dane do własnych projektów analitycznych i uczenia maszynowego.

Przykład analizy danych – studium przypadku krok po kroku

Analiza danych (EDA – Exploratory Data Analysis) to pierwszy i najważniejszy etap pracy z danymi. Zanim zbudujemy model uczenia maszynowego, musimy zrozumieć, co jest w zbiorze: jakie mamy kolumny, jakie są wartości, czy są braki danych, czy dane są poprawnie wczytane, oraz jak wygląda zmienna docelowa (czyli to, co chcemy przewidywać). W praktyce EDA odpowiada na pytania: *„czy dane są dobre?”*, *„czy są błędy?”*, *„co wyróżnia osoby z klasą 1 od klasy 0?”*.

W tym studium przypadku analizujemy dane BRFSS 2015 związane ze zdrowiem oraz cukrzycą – dane pochodzą ze wspominanego wcześniej portalu Kaggle. Zmienna Diabetes_binary opisuje, czy dana osoba ma cukrzycę (1) czy nie (0). W analizie wykonamy: wczytanie danych, podstawową kontrolę jakości (braki, duplikaty), statystyki (średnie, mediany, odchylenia), rozkłady cech (histogramy), porównania cech między klasami (np. cukrzyca vs brak), wykresy dla zmiennych binarnych (np. czy ktoś pali), wizualizację korelacji oraz prostą sekcję wniosków. To będzie wzorzec, jak robi się analizę danych w Pythonie krok po kroku.

```
# ============================================================
# 0) IMPORTY I USTAWIENIA (zawsze na początku notebooka)
# ============================================================

import pandas as pd  # biblioteka do pracy z tabelami (DataFrame)
import numpy as np  # biblioteka do obliczeń matematycznych
import matplotlib.pyplot as plt  # biblioteka do tworzenia wykresów
import seaborn as sns  # biblioteka do ładniejszych wykresów statystycznych
from pathlib import Path  # wygodna praca ze ścieżkami plików

pd.set_option("display.max_columns", 200)  # pokazuj dużo kolumn w tabelach
pd.set_option("display.width", 140)  # ustaw szerokość wyświetlania tabel
sns.set_theme(style="whitegrid")  # ustaw przyjemny styl wykresów

RANDOM_STATE = 42  # stałe ziarno losowości (dla powtarzalności)
SAMPLE_N = 8000  # próbka do cięższych wykresów (żeby nie liczyć na 250k
wierszy)
```

```
# ============================================================
# 1) WCZYTANIE DANYCH
# ============================================================

FILE_PATH = Path("diabetes_binary_health_indicators_BRFSS2015 (3) (1) (1).csv")  # nazwa pliku danych
df = pd.read_csv(FILE_PATH)  # wczytaj plik CSV do DataFrame

print("Wymiary danych (wiersze, kolumny):", df.shape)  # pokaż rozmiar danych
df.head(10)  # pokaż pierwsze 10 wierszy (szybki podgląd)

# ============================================================
# 2) SPRAWDZENIE STRUKTURY DANYCH (typy, kolumny, podstawy)
# ============================================================

print("\nNazwy kolumn:\n", list(df.columns))  # wypisz nazwy kolumn
df.info()  # pokaż typy danych i liczbę niepustych wartości w kolumnach
df.describe().T  # pokaż podstawowe statystyki liczbowe (transponowane)

# ============================================================
# 3) BRAKI DANYCH I DUPLIKATY (kontrola jakości)
# ============================================================

missing_count = df.isna().sum()  # policz braki danych w każdej kolumnie
missing_percent = (missing_count / len(df)) * 100  # policz procent braków w każdej kolumnie

missing_summary = pd.DataFrame({  # zbuduj tabelę podsumowania braków
    "missing_count": missing_count,  # liczba braków
    "missing_percent": missing_percent  # procent braków
}).sort_values("missing_count", ascending=False)  # posortuj od największej liczby braków

print("\nBraki danych (top 10 kolumn):")  # opis
```

```
display(missing_summary.head(10))  # pokaż 10 kolumn z największą liczbą braków

duplicates_n = df.duplicated().sum()  # policz liczbę zduplikowanych wierszy
print("\nLiczba duplikatów wierszy:", duplicates_n)  # wyświetl wynik

# Uwaga: w EDA zwykle nie usuwamy nic od razu, tylko najpierw rozumiemy dane.  # komentarz dydaktyczny

# ============================================================
# 4) ZMIENNA DOCELOWA (co chcemy analizować / przewidywać)
# ============================================================

target_col = "Diabetes_binary"  # nazwa zmiennej docelowej (0 = brak, 1 = cukrzyca)

target_counts = df[target_col].value_counts(dropna=False)  # policz liczbę przypadków w klasach
target_percent = df[target_col].value_counts(normalize=True) * 100  # policz procent klas

print("\nRozkład klas (liczności):\n", target_counts)  # pokaż liczności
print("\nRozkład klas (procent):\n", target_percent.round(2))  # pokaż procenty

plt.figure(figsize=(6, 4))  # ustaw rozmiar wykresu
sns.countplot(data=df, x=target_col)  # wykres liczności klas
plt.title("Rozkład klasy docelowej: cukrzyca (1) vs brak (0)")  # tytuł
plt.xlabel("Diabetes_binary")  # podpis osi X
plt.ylabel("Liczba obserwacji")  # podpis osi Y
plt.show()  # pokaż wykres

# ============================================================
# 5) PODZIAŁ KOLUMN NA GRUPY (żeby analizować mądrze)
# ============================================================
```

```
"""
W tych danych większość kolumn jest binarna (0/1).
Są też kolumny porządkowe (np. Age, GenHlth, Education, Income),
oraz liczbowe (np. BMI, MentHlth, PhysHlth).

Ten podział pomaga dobrać odpowiedni wykres i sposób analizy.
"""

binary_cols = [  # kolumny binarne (0/1)
    "HighBP", "HighChol", "CholCheck", "Smoker", "Stroke",
"HeartDiseaseorAttack",
    "PhysActivity", "Fruits", "Veggies", "HvyAlcoholConsump",
"AnyHealthcare",
    "NoDocbcCost", "DiffWalk", "Sex"
]  # koniec listy binarnej

ordinal_cols = ["GenHlth", "Age", "Education", "Income"]  # kolumny
porządkowe (liczby oznaczają kategorie)
numeric_cols = ["BMI", "MentHlth", "PhysHlth"]  # kolumny liczbowe (ciągłe
lub dyskretne)

all_used = set(binary_cols + ordinal_cols + numeric_cols + [target_col])  #
zbiór wszystkich użytych kolumn
unused_cols = [c for c in df.columns if c not in all_used]  # kolumny
nieuwzględnione w podziale

print("\nKolumny nieuwzględnione w podziale (sprawdź czy chcesz je
dodać):", unused_cols)  # pokaż ewentualne braki

# ============================================================
# 6) MIARY STATYSTYCZNE (średnia, mediana, odchylenie, kwartyle, IQR)
# ============================================================
```

```
"""
Miary statystyczne pomagają opisać dane liczbowo:
- średnia i mediana mówią o "typowej" wartości,
- odchylenie standardowe mówi, jak bardzo dane są rozproszone,
- kwartyle i IQR pomagają rozumieć rozkład bez wrażliwości na skrajności.
"""

desc_num = df[numeric_cols].describe().T  # podstawowe statystyki (count, mean, std, min, Q1, Q2, Q3, max)
desc_num["median"] = df[numeric_cols].median()  # dodaj medianę
desc_num["skew"] = df[numeric_cols].skew()  # dodaj skośność
desc_num["kurtosis"] = df[numeric_cols].kurtosis()  # dodaj kurtozę

q1 = df[numeric_cols].quantile(0.25)  # pierwszy kwartyl
q3 = df[numeric_cols].quantile(0.75)  # trzeci kwartyl
iqr = q3 - q1  # rozstęp międzykwartylowy

desc_num["Q1"] = q1  # dodaj Q1
desc_num["Q3"] = q3  # dodaj Q3
desc_num["IQR"] = iqr  # dodaj IQR

print("\nStatystyki dla kolumn liczbowych:")  # opis
display(desc_num)  # pokaż tabelę

# ============================================================
# 7) ROZKŁADY ZMIENNYCH LICZBOWYCH (histogram + KDE)
# ============================================================

for col in numeric_cols:  # przejdź po każdej kolumnie liczbowej
    plt.figure(figsize=(7, 4))  # ustaw rozmiar wykresu
    sns.histplot(df[col], bins=40, kde=True)  # histogram z linią KDE
    plt.title(f"Rozkład zmiennej: {col}")  # tytuł
    plt.xlabel(col)  # podpis osi X
    plt.ylabel("Liczba obserwacji")  # podpis osi Y
    plt.show()  # pokaż wykres
```

```
# ============================================================
# 8) OUTLIERY (wartości odstające) – boxploty dla liczb
# ============================================================

for col in numeric_cols:  # iteruj po kolumnach liczbowych
    plt.figure(figsize=(7, 2.5))  # rozmiar wykresu
    sns.boxplot(x=df[col])  # boxplot pokazuje medianę, kwartyle i wartości odstające
    plt.title(f"Wartości odstające (boxplot): {col}")  # tytuł
    plt.xlabel(col)  # podpis osi
    plt.show()  # pokaż wykres

# ============================================================
# 9) PORÓWNANIE LICZBOWYCH CECH DLA KLAS (0 vs 1)
# ============================================================

"""
Tutaj sprawdzamy: czy np. BMI różni się między osobami z cukrzycą i bez cukrzycy?
To jest kluczowy element EDA pod ML: "co odróżnia klasy?"
"""

for col in numeric_cols:  # iteruj po cechach liczbowych
    plt.figure(figsize=(7, 4))  # rozmiar wykresu
    sns.boxplot(data=df, x=target_col, y=col)  # boxplot w podziale na klasy 0/1
    plt.title(f"{col} w podziale na klasy (Diabetes_binary)")  # tytuł
    plt.xlabel("Diabetes_binary (0=brak, 1=cukrzyca)")  # opis osi X
    plt.ylabel(col)  # opis osi Y
    plt.show()  # pokaż wykres

# Dodatkowe zestawienie liczbowe (średnie w klasach)  # komentarz dydaktyczny
means_by_class = df.groupby(target_col)[numeric_cols].mean()  # policz średnie cech liczbowych w klasach
```

```
std_by_class = df.groupby(target_col)[numeric_cols].std()  # policz
odchylenia w klasach

print("\nŚrednie cech liczbowych w klasach:")  # opis
display(means_by_class)  # pokaż średnie

print("\nOdchylenia standardowe cech liczbowych w klasach:")  # opis
display(std_by_class)  # pokaż odchylenia

# ============================================================
# 10) ZMIENNE BINARNE (0/1) - rozkład i związek z cukrzycą
# ============================================================

"""
Dla zmiennych 0/1 najlepszy jest:
- countplot (ile jest 0 i 1),
- wykres udziału (procent),
- porównanie z targetem (np. odsetek cukrzycy w grupie 0 vs 1).
"""

for col in binary_cols:  # przejdź po każdej zmiennej binarnej
    plt.figure(figsize=(7, 4))  # rozmiar wykresu
    sns.countplot(data=df, x=col, hue=target_col)  # rozkład 0/1 i
jednocześnie klasy docelowej
    plt.title(f"{col} vs Diabetes_binary (liczności)")  # tytuł
    plt.xlabel(col)  # oś X
    plt.ylabel("Liczba obserwacji")  # oś Y
    plt.legend(title="Diabetes_binary")  # legenda
    plt.show()  # pokaż wykres

# Tabela: jaki procent cukrzycy jest w grupie 0 i 1 dla każdej cechy
binarnej  # komentarz dydaktyczny
binary_target_rates = {}  # słownik na wyniki
```

```
for col in binary_cols:  # iteruj po cechach binarnych
    rate = df.groupby(col)[target_col].mean() * 100  # średnia z 0/1 daje odsetek (w %)
    binary_target_rates[col] = rate  # zapisz do słownika

binary_target_rates_df = pd.DataFrame(binary_target_rates).T  # zamień słownik na tabelę
binary_target_rates_df.columns = ["% cukrzycy gdy cecha=0", "% cukrzycy gdy cecha=1"]  # nadaj nazwy kolumn

print("\nOdsetek cukrzycy w grupach 0/1 dla zmiennych binarnych:")  # opis
display(binary_target_rates_df.sort_values("% cukrzycy gdy cecha=1", ascending=False))  # pokaż posortowane

# ============================================================
# 11) ZMIENNE PORZĄDKOWE - rozkłady i związek z targetem
# ============================================================

for col in ordinal_cols:  # iteruj po zmiennych porządkowych
    plt.figure(figsize=(7, 4))  # rozmiar
    sns.countplot(data=df, x=col, hue=target_col)  # rozkład kategorii i podział na target
    plt.title(f"{col} vs Diabetes_binary (liczności)")  # tytuł
    plt.xlabel(col)  # oś X
    plt.ylabel("Liczba obserwacji")  # oś Y
    plt.legend(title="Diabetes_binary")  # legenda
    plt.show()  # pokaż

# Średni odsetek cukrzycy w zależności od kategorii porządkowej  # komentarz dydaktyczny
for col in ordinal_cols:  # iteruj po zmiennych porządkowych
    rate_by_level = df.groupby(col)[target_col].mean() * 100  # odsetek cukrzycy na poziomach
    print(f"\nOdsetek cukrzycy (%) w zależności od {col}:")  # opis
    display(rate_by_level)  # pokaż
```

```
# ============================================================
# 12) KORELACJE (heatmapa + lista najmocniejszych)
# ============================================================

"""
Korelacja pokazuje, jak silnie zmienne są ze sobą powiązane liniowo.
Warto pamiętać: korelacja nie oznacza przyczynowości.
To jednak świetny szybki podgląd zależności w danych.
"""

corr = df.corr(numeric_only=True)  # policz macierz korelacji (tylko liczby)

plt.figure(figsize=(12, 9))  # rozmiar wykresu
sns.heatmap(corr, cmap="coolwarm", center=0, square=True)  # heatmapa korelacji
plt.title("Macierz korelacji (heatmapa)")  # tytuł
plt.show()  # pokaż

# Najmocniejsze korelacje ze zmienną docelową  # komentarz dydaktyczny
corr_with_target = corr[target_col].sort_values(ascending=False)  # sortuj korelacje z targetem
print("\nKorelacje z targetem (od największej):")  # opis
display(corr_with_target)  # pokaż

# ============================================================
# 13) PAIRPLOT (opcjonalnie) - na próbce danych (bo pełny zbiór jest duży)
# ============================================================

"""
Pairplot jest bardzo fajny, ale kosztowny obliczeniowo.
Dlatego robimy go na losowej próbce danych.
"""

df_sample = df.sample(n=min(SAMPLE_N, len(df)),
random_state=RANDOM_STATE)  # weź próbkę danych
```

```
sns.pairplot(df_sample[[target_col] + numeric_cols], hue=target_col,
diag_kind="hist")  # pairplot na cechach liczbowych
plt.show()  # pokaż

# ============================================================
# 14) PODSUMOWANIE EDA – wnioski (szablon do uzupełnienia)
# ============================================================

"""
W tym miejscu (w skrypcie) warto dopisać 5–10 zdań wniosków, np.:
- czy klasy są zbalansowane?
- które cechy najbardziej różnią się między 0 i 1?
- czy są outliery w BMI / PhysHlth / MentHlth?
- jakie cechy mają najwyższą korelację z targetem?
- jakie problemy jakości danych zauważyliśmy?
"""
```

Przykład budowy modelu uczenia maszynowego – regresja logistyczna krok po kroku

W niniejszym podrozdziale przedstawiony został kompletny, praktyczny przykład budowy modelu uczenia maszynowego w języku Python, z wykorzystaniem regresji logistycznej. Celem tego przykładu nie jest uzyskanie najlepszego możliwego modelu predykcyjnego, lecz zrozumienie całego procesu tworzenia modelu ML, od przygotowanych wcześniej danych aż do użycia wytrenowanego modelu do predykcji dla nowych przypadków. Regresja logistyczna została wybrana jako pierwszy model, ponieważ jest stosunkowo prosta, dobrze interpretowalna i często wykorzystywana jako punkt odniesienia w projektach analitycznych.

W ramach przykładu pokazano wszystkie kluczowe etapy pracy z modelem uczenia maszynowego. Dane zostały podzielone na zbiory treningowe i testowe, z zachowaniem proporcji klas, aby zapewnić rzetelną ocenę jakości modelu. Następnie zastosowano balansowanie klas metodą SMOTE, co pozwala lepiej radzić sobie z problemem niezbalansowanych danych, typowym dla zagadnień medycznych. Kolejnym krokiem było

skalowanie cech, które jest istotne w przypadku regresji logistycznej i wpływa na stabilność procesu uczenia.

W podrozdziale zaprezentowano również ocenę jakości modelu przy użyciu najważniejszych metryk klasyfikacji, takich jak accuracy, precision, recall oraz F1-score, a także wizualizacje w postaci macierzy pomyłek oraz krzywych ROC i Precision–Recall. Szczególną uwagę poświęcono świadomemu wyborowi progu decyzyjnego, pokazując, że domyślna wartość 0.5 nie zawsze jest najlepszym rozwiązaniem, zwłaszcza w problemach związanych ze zdrowiem.

Na końcu przedstawiono praktyczny aspekt wykorzystania modelu: zapis wytrenowanego modelu do pliku, jego ponowne wczytanie oraz użycie do predykcji dla danych wprowadzonych przez użytkownika. Dzięki temu przykład ten stanowi pełne studium przypadku uczenia maszynowego, pokazujące, jak teoria i narzędzia Pythona przekładają się na rzeczywisty proces budowy i użycia modelu.

```
# ============================================================
# WZORCOWY PROJEKT ML (Regresja Logistyczna) – DIABETES BRFSS 2015
# ============================================================
# Cel:
# 1) Wczytać dane (CSV)
# 2) Przygotować X i y
# 3) Podzielić dane na train i test
# 4) Zbalansować tylko TRAIN (SMOTE)
# 5) Zeskalować dane (StandardScaler)
# 6) Wytrenować LogisticRegression
# 7) Sprawdzić metryki i zrobić wykresy (CM, ROC, PR)
# 8) Zapisać model do pliku i wczytać go z powrotem
# 9) Zapytać użytkownika o parametry i zdiagnozować cukrzycę
# ============================================================


# --------------------------
# 0) INSTALACJE (tylko w Colabie, jeśli brakuje pakietu)
# --------------------------
```

```
# Jeśli w Colabie nie masz imbalanced-learn, odkomentuj tę linię.  # komentarz
# !pip -q install imbalanced-learn  # instalacja biblioteki do SMOTE (balansowanie)


# ---------------------------
# 1) IMPORTY
# ---------------------------

import pandas as pd  # praca z danymi tabelarycznymi (DataFrame)
import numpy as np  # obliczenia numeryczne
import matplotlib.pyplot as plt  # wykresy
import seaborn as sns  # wykresy statystyczne
from pathlib import Path  # ścieżki do plików

from sklearn.model_selection import train_test_split  # podział danych na train/test
from sklearn.preprocessing import StandardScaler  # skalowanie cech
from sklearn.linear_model import LogisticRegression  # model regresji logistycznej

from sklearn.metrics import accuracy_score  # accuracy
from sklearn.metrics import precision_score  # precision
from sklearn.metrics import recall_score  # recall
from sklearn.metrics import f1_score  # F1
from sklearn.metrics import confusion_matrix  # macierz pomyłek
from sklearn.metrics import classification_report  # raport klasyfikacji
from sklearn.metrics import roc_curve  # punkty krzywej ROC
from sklearn.metrics import roc_auc_score  # AUC
from sklearn.metrics import precision_recall_curve  # krzywa precision-recall
from sklearn.metrics import average_precision_score  # AP (pole pod PR)

from imblearn.over_sampling import SMOTE  # SMOTE do balansowania klas
```

```
import pickle  # zapis/wczytanie modelu jako plik binarny

sns.set_theme(style="whitegrid")  # ładny styl wykresów


# ----------------------------
# 2) WCZYTANIE DANYCH
# ----------------------------

FILE_PATH = Path("diabetes_binary_health_indicators_BRFSS2015 (3) (1) (1).csv")  # ścieżka do pliku CSV
df = pd.read_csv(FILE_PATH)  # wczytaj dane z CSV do DataFrame

print("Wymiary danych (wiersze, kolumny):", df.shape)  # szybka informacja o rozmiarze danych
print("Pierwsze 3 wiersze danych:")  # opis
display(df.head(3))  # podgląd danych


# ----------------------------
# 3) MINIMALNA KONTROLA JAKOŚCI (bez powtarzania pełnej EDA)
# ----------------------------

"""
W EDA sprawdzaliśmy dane dokładniej.
Tutaj robimy tylko krótką kontrolę, żeby modelowanie było bezpieczne:
- czy są braki danych?
- czy target ma tylko 0/1?
- czy typy są liczbowe?
"""

print("\nBraki danych w całym zbiorze (suma):", df.isna().sum().sum())  # suma braków w całym zbiorze
print("Unikalne wartości w Diabetes_binary:", df["Diabetes_binary"].unique())  # sprawdź klasy 0/1
```

```
# ---------------------------
# 4) WYBÓR ZMIENNEJ DOCELOWEJ I CECH
# ---------------------------

target_col = "Diabetes_binary"  # nazwa kolumny, którą chcemy przewidywać (0/1)

X = df.drop(columns=[target_col])  # X = cechy (wszystko oprócz targetu)
y = df[target_col]  # y = target (0/1)

feature_names = list(X.columns)  # zapamiętaj nazwy cech (ważne do eksportu i późniejszego użycia)

print("\nLiczba cech (kolumn wejściowych):", X.shape[1])  # ile cech mamy w modelu
print("Nazwy cech:", feature_names)  # pokaż nazwy cech


# ---------------------------
# 5) PODZIAŁ NA TRAIN / TEST
# ---------------------------

"""
To jest kluczowa zasada:
- model uczy się tylko na TRAIN
- ocena jakości jest na TEST (danych niewidzianych)

Używamy stratify=y, aby proporcje klas były podobne w train i test.
"""

X_train, X_test, y_train, y_test = train_test_split(  # wykonaj podział
    X,  # cechy
    y,  # target
    test_size=0.2,  # 20% danych na test
    random_state=42,  # powtarzalność
    stratify=y  # zachowaj proporcje klas
```

```
)

print("\nRozmiary zbiorów:")  # opis
print("X_train:", X_train.shape, "X_test:", X_test.shape)  # pokaż wymiary
print("y_train:", y_train.shape, "y_test:", y_test.shape)  # pokaż wymiary


# ----------------------------
# 6) BALANSOWANIE KLAS (SMOTE) - TYLKO NA TRAIN
# ----------------------------

"""
Dane są niezbalansowane (zwykle więcej osób bez cukrzycy niż z cukrzycą).
SMOTE tworzy „syntetyczne” przykłady klasy mniejszościowej.
UWAGA: SMOTE robimy TYLKO na TRAIN, bo inaczej „przeciekamy” informacją do
testu.
"""

smote = SMOTE(random_state=42)  # utwórz obiekt SMOTE
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)  #
zbalansuj train

print("\nRozkład klas przed SMOTE (train):")  # opis
print(y_train.value_counts())  # ile było klas w train

print("\nRozkład klas po SMOTE (train):")  # opis
print(pd.Series(y_train_res).value_counts())  # ile jest klas po SMOTE


# ----------------------------
# 7) SKALOWANIE DANYCH (StandardScaler)
# ----------------------------

"""
Regresja logistyczna działa lepiej, gdy cechy mają podobną skalę.
StandardScaler robi:
```

```
- odejmuje średnią
- dzieli przez odchylenie standardowe

BARDZO WAŻNE:
- scaler FIT na TRAIN (po SMOTE)
- scaler TRANSFORM na TRAIN i TEST
"""

scaler = StandardScaler()  # utwórz scaler

X_train_scaled = scaler.fit_transform(X_train_res)  # dopasuj scaler do
train i przeskaluj train
X_test_scaled = scaler.transform(X_test)  # przeskaluj test tym samym
scalerem (bez fit!)

print("\nWymiary po skalowaniu:")  # opis
print("X_train_scaled:", X_train_scaled.shape)  # wymiary train
print("X_test_scaled:", X_test_scaled.shape)  # wymiary test


# ---------------------------
# 8) TRENING MODELU: REGRESJA LOGISTYCZNA
# ---------------------------

"""
Regresja logistyczna:
- uczy się zależności między cechami a klasą 0/1
- zwraca też prawdopodobieństwo (predict_proba)

Ustawiamy max_iter=1000, aby model miał czas „dojść do rozwiązania”.
"""

model = LogisticRegression(max_iter=1000, random_state=42)  # utwórz model
model.fit(X_train_scaled, y_train_res)  # naucz model na zbalansowanym
train
```

```
# ----------------------------
# 9) PREDYKCJA (TEST) + PRAWDOPODOBIEŃSTWA
# ----------------------------

y_pred = model.predict(X_test_scaled)  # przewidywana klasa (0/1)
y_proba = model.predict_proba(X_test_scaled)[:, 1]  # prawdopodobieństwo
klasy 1 (cukrzyca)


# ----------------------------
# 10) METRYKI JAKOŚCI MODELU
# ----------------------------

"""
W medycznych przykładach szczególnie ważny bywa recall (czułość):
- ile prawdziwych przypadków cukrzycy wykryliśmy?

Ale pokazujemy pełen pakiet metryk:
- accuracy, precision, recall, f1
- confusion matrix
- classification report
"""

acc = accuracy_score(y_test, y_pred)  # accuracy
prec = precision_score(y_test, y_pred, zero_division=0)  # precision
rec = recall_score(y_test, y_pred, zero_division=0)  # recall
f1 = f1_score(y_test, y_pred, zero_division=0)  # f1

print("\n=== METRYKI (na zbiorze testowym) ===")  # nagłówek
print(f"Accuracy : {acc:.4f}")  # accuracy
print(f"Precision: {prec:.4f}")  # precision
print(f"Recall   : {rec:.4f}")  # recall
print(f"F1-score : {f1:.4f}")  # f1

print("\n=== RAPORT KLASYFIKACJI ===")  # nagłówek
```

```
print(classification_report(y_test, y_pred, digits=4))  # raport
klasyfikacji


# ---------------------------
# 11) MACIERZ POMYŁEK + HEATMAPA
# ---------------------------

cm = confusion_matrix(y_test, y_pred)  # policz macierz pomyłek

plt.figure(figsize=(7, 5))  # rozmiar wykresu
sns.heatmap(  # heatmapa
    cm,  # macierz
    annot=True,  # pokaż liczby
    fmt="d",  # format liczb (integer)
    cmap="Blues",  # kolory
    xticklabels=["brak cukrzycy (0)", "cukrzyca (1)"],  # etykiety osi X
    yticklabels=["brak cukrzycy (0)", "cukrzyca (1)"]  # etykiety osi Y
)
plt.title("Macierz pomyłek (Confusion Matrix)")  # tytuł
plt.xlabel("Klasa przewidziana")  # oś X
plt.ylabel("Klasa rzeczywista")  # oś Y
plt.show()  # pokaż wykres


# ---------------------------
# 12) KRZYWA ROC + AUC
# ---------------------------

auc = roc_auc_score(y_test, y_proba)  # policz AUC
fpr, tpr, thresholds = roc_curve(y_test, y_proba)  # punkty ROC

plt.figure(figsize=(7, 5))  # rozmiar
plt.plot(fpr, tpr, lw=2, label=f"ROC (AUC = {auc:.4f})")  # krzywa ROC
plt.plot([0, 1], [0, 1], linestyle="--", lw=2, label="Losowy
klasyfikator")  # przekątna
```

```
plt.title("Krzywa ROC")  # tytuł
plt.xlabel("False Positive Rate (FPR)")  # oś X
plt.ylabel("True Positive Rate (TPR)")  # oś Y
plt.legend()  # legenda
plt.show()  # pokaż


# ---------------------------
# 13) KRZYWA PRECISION-RECALL + AVERAGE PRECISION (AP)
# ---------------------------

precision, recall, pr_thresholds = precision_recall_curve(y_test,
y_proba)  # policz PR curve
ap = average_precision_score(y_test, y_proba)  # pole pod krzywą PR

plt.figure(figsize=(7, 5))  # rozmiar
plt.plot(recall, precision, lw=2, label=f"PR curve (AP = {ap:.4f})")  #
wykres PR
plt.title("Krzywa Precision-Recall")  # tytuł
plt.xlabel("Recall")  # oś X
plt.ylabel("Precision")  # oś Y
plt.legend()  # legenda
plt.show()  # pokaż


# ---------------------------
# 14) ROZKŁAD PRAWDOPODOBIEŃSTW (czy model rozróżnia klasy?)
# ---------------------------

plt.figure(figsize=(8, 5))  # rozmiar
sns.histplot(y_proba[y_test == 0], bins=40, kde=True, label="Brak cukrzycy
(0)")  # rozkład proba dla klasy 0
sns.histplot(y_proba[y_test == 1], bins=40, kde=True, label="Cukrzyca
(1)")  # rozkład proba dla klasy 1
plt.title("Rozkład przewidywanego prawdopodobieństwa klasy 1")  # tytuł
plt.xlabel("P(y=1) = prawdopodobieństwo cukrzycy")  # oś X
```

```
plt.ylabel("Liczba przypadków")  # oś Y
plt.legend()  # legenda
plt.show()  # pokaż


# ---------------------------
# 15) ZAPIS MODELU DO PLIKU (eksport)
# ---------------------------

"""
W praktyce nie zapisujemy tylko samego modelu.
Musimy zapisać też:
- scaler (bo bez niego nowe dane będą w innej skali)
- listę cech i ich kolejność (żeby użytkownik podawał dane w tej samej kolejności)
"""

artifact = {  # tworzymy „paczkę” do zapisu
    "model": model,  # wytrenowany model
    "scaler": scaler,  # dopasowany scaler
    "feature_names": feature_names  # kolejność cech
}

MODEL_FILE = "logreg_diabetes_model.pkl"  # nazwa pliku binarnego

with open(MODEL_FILE, "wb") as f:  # otwórz plik do zapisu binarnego
    pickle.dump(artifact, f)  # zapisz paczkę do pliku

print("\nModel zapisany do pliku:", MODEL_FILE)  # potwierdzenie


# ---------------------------
# 16) WCZYTANIE MODELU (import) – test, czy wszystko działa
# ---------------------------

with open(MODEL_FILE, "rb") as f:  # otwórz plik binarny do odczytu
```

```
    loaded_artifact = pickle.load(f)  # wczytaj paczkę

loaded_model = loaded_artifact["model"]  # wyciągnij model
loaded_scaler = loaded_artifact["scaler"]  # wyciągnij scaler
loaded_feature_names = loaded_artifact["feature_names"]  # wyciągnij listę
cech

# Krótki test: predykcja na pierwszych 5 rekordach testu  # komentarz
test_scaled_again = loaded_scaler.transform(X_test)  # przeskaluj test tak
jak wcześniej
test_pred_again = loaded_model.predict(test_scaled_again[:5])  # przewidź
klasy dla 5 pierwszych

print("\nTest po imporcie (pierwsze 5 predykcji):", test_pred_again)  #
pokaż wynik


# ----------------------------
# 17) „DIAGNOZA” - zapytaj użytkownika o cechy i przewidź cukrzycę
# ----------------------------

"""
To jest prosty „interfejs tekstowy”.
Użytkownik wpisuje wartości cech (tak jak w danych), a my:
1) tworzymy wiersz danych w odpowiedniej kolejności
2) skalujemy go scalerem
3) liczymy predykcję i prawdopodobieństwo

WAŻNE: To jest przykład edukacyjny, nie narzędzie medyczne.
"""

def ask_float(name, hint):  # funkcja pomocnicza do wczytywania liczb
    value = float(input(f"{name} ({hint}): "))  # pobierz wartość od
użytkownika
    return value  # zwróć wartość
```

```
print("\n=== WPROWADŹ DANE PACJENTA (jak w zbiorze) ===")  # nagłówek

user_data = {}  # słownik na dane od użytkownika

# Każda linia poniżej pyta o jedną cechę.  # komentarz
user_data["HighBP"] = ask_float("HighBP", "0 lub 1 (wysokie ciśnienie)")  # HighBP
user_data["HighChol"] = ask_float("HighChol", "0 lub 1 (wysoki cholesterol)")  # HighChol
user_data["CholCheck"] = ask_float("CholCheck", "0 lub 1 (badanie cholesterolu)")  # CholCheck
user_data["BMI"] = ask_float("BMI", "np. 18–50 (wskaźnik BMI)")  # BMI
user_data["Smoker"] = ask_float("Smoker", "0 lub 1 (czy palił >=100 papierosów)")  # Smoker
user_data["Stroke"] = ask_float("Stroke", "0 lub 1 (czy miał udar)")  # Stroke
user_data["HeartDiseaseorAttack"] = ask_float("HeartDiseaseorAttack", "0 lub 1 (choroba serca/zawał)")  # HeartDiseaseorAttack
user_data["PhysActivity"] = ask_float("PhysActivity", "0 lub 1 (aktywność fizyczna)")  # PhysActivity
user_data["Fruits"] = ask_float("Fruits", "0 lub 1 (czy je owoce)")  # Fruits
user_data["Veggies"] = ask_float("Veggies", "0 lub 1 (czy je warzywa)")  # Veggies
user_data["HvyAlcoholConsump"] = ask_float("HvyAlcoholConsump", "0 lub 1 (duże spożycie alkoholu)")  # HvyAlcoholConsump
user_data["AnyHealthcare"] = ask_float("AnyHealthcare", "0 lub 1 (ubezpieczenie/opieka)")  # AnyHealthcare
user_data["NoDocbcCost"] = ask_float("NoDocbcCost", "0 lub 1 (czy koszt blokował wizytę)")  # NoDocbcCost
user_data["GenHlth"] = ask_float("GenHlth", "1–5 (ogólny stan zdrowia)")  # GenHlth
user_data["MentHlth"] = ask_float("MentHlth", "0–30 (dni gorszego zdrowia psych.)")  # MentHlth
user_data["PhysHlth"] = ask_float("PhysHlth", "0–30 (dni gorszego zdrowia fiz.)")  # PhysHlth
```

```
user_data["DiffWalk"] = ask_float("DiffWalk", "0 lub 1 (trudność w chodzeniu)")  # DiffWalk
user_data["Sex"] = ask_float("Sex", "0 lub 1 (płeć jak w danych)")  # Sex
user_data["Age"] = ask_float("Age", "1-13 (przedziały wieku jak w danych)")  # Age
user_data["Education"] = ask_float("Education", "1-6 (poziom edukacji)")  # Education
user_data["Income"] = ask_float("Income", "1-8 (przedziały dochodu)")  # Income

# Zbuduj DataFrame z jednym wierszem, w poprawnej kolejności cech.  # komentarz
user_df = pd.DataFrame([user_data], columns=loaded_feature_names)  # jedna obserwacja jako DataFrame

# Skaluj dane tak jak trenowaliśmy model.  # komentarz
user_scaled = loaded_scaler.transform(user_df)  # przeskaluj dane użytkownika

# Przewidź klasę i prawdopodobieństwo.  # komentarz
user_pred = loaded_model.predict(user_scaled)[0]  # przewidywana klasa (0/1)
user_prob = loaded_model.predict_proba(user_scaled)[0, 1]  # prawdopodobieństwo klasy 1

print("\n=== WYNIK MODELU ===")  # nagłówek
print("Predykcja (0=brak cukrzycy, 1=cukrzyca):", int(user_pred))  # pokaż klasę
print(f"Prawdopodobieństwo cukrzycy (klasa 1): {user_prob:.4f}")  # pokaż prawdopodobieństwo

# Dodatkowy komunikat dla czytelności
if user_pred == 1:  # jeśli model przewiduje cukrzycę
    print("Model sugeruje: WYSOKIE RYZYKO (klasa 1).")  # komunikat
else:  # jeśli model przewiduje brak cukrzycy
    print("Model sugeruje: NISKIE RYZYKO (klasa 0).")  # komunikat
```

```
# ============================================================
# 18) ZMIANA PROGU DECYZYJNEGO (threshold)
# ============================================================

"""
Domyślnie klasyfikator przewiduje klasę 1, gdy prawdopodobieństwo >= 0.50.
W praktyce (zwłaszcza w medycynie) często chcemy:
- zwiększyć recall (wykryć więcej chorych),
kosztem precision (więcej fałszywych alarmów).

Dlatego uczymy się wybierać próg (threshold) świadomie.
"""

# ---------------------------
# 18.1) Obliczamy precision i recall dla różnych progów
# ---------------------------

precisions, recalls, thr = precision_recall_curve(y_test, y_proba)  #
policz PR dla wielu progów

thr_safe = np.append(thr, 1.0)  # dodaj próg 1.0, żeby długości tablic
pasowały (thr jest o 1 krótsze)

# ---------------------------
# 18.2) Wykres: precision i recall w zależności od progu
# ---------------------------

plt.figure(figsize=(8, 5))  # rozmiar wykresu
plt.plot(thr_safe, precisions, label="Precision")  # precision vs threshold
plt.plot(thr_safe, recalls, label="Recall")  # recall vs threshold
plt.title("Precision i Recall w zależności od progu (threshold)")  # tytuł
plt.xlabel("Threshold (próg decyzji)")  # oś X
plt.ylabel("Wartość metryki")  # oś Y
plt.legend()  # legenda
plt.show()  # pokaż wykres
```

```
# ---------------------------
# 18.3) Prosty wybór progu: maksymalizacja F1
# ---------------------------

"""
Wybór progu można robić na wiele sposobów.
Najprostszy i bardzo edukacyjny wariant:
- wybieramy threshold, który daje najwyższy F1 na zbiorze testowym.

Uwaga dydaktyczna:
W idealnym świecie próg wybieramy na walidacji, a test zostawiamy „na koniec”.
Tutaj robimy to prosto, żeby zrozumieć ideę.
"""

f1_scores = (2 * precisions * recalls) / (precisions + recalls + 1e-12)  # oblicz F1 dla każdego punktu (bez dzielenia przez zero)
best_idx = np.argmax(f1_scores)  # indeks najlepszego F1
best_threshold = thr_safe[best_idx]  # próg odpowiadający najlepszemu F1

print("\n=== WYBÓR PROGU (na podstawie najlepszego F1) ===")  # nagłówek
print("Najlepszy threshold:", round(float(best_threshold), 4))  # pokaż próg
print("F1 dla tego progu:", round(float(f1_scores[best_idx]), 4))  # pokaż F1

# ---------------------------
# 18.4) Predykcja z nowym progiem + metryki
# ---------------------------

y_pred_thr = (y_proba >= best_threshold).astype(int)  # zamień prawdopodobieństwa na klasy używając nowego progu

acc_thr = accuracy_score(y_test, y_pred_thr)  # accuracy dla progu
```

```
prec_thr = precision_score(y_test, y_pred_thr, zero_division=0)  # precision dla progu
rec_thr = recall_score(y_test, y_pred_thr, zero_division=0)  # recall dla progu
f1_thr = f1_score(y_test, y_pred_thr, zero_division=0)  # f1 dla progu

print("\n=== METRYKI DLA NOWEGO PROGU ===")  # nagłówek
print(f"Accuracy : {acc_thr:.4f}")  # accuracy
print(f"Precision: {prec_thr:.4f}")  # precision
print(f"Recall   : {rec_thr:.4f}")  # recall
print(f"F1-score : {f1_thr:.4f}")  # f1

print("\n=== RAPORT KLASYFIKACJI DLA NOWEGO PROGU ===")  # nagłówek
print(classification_report(y_test, y_pred_thr, digits=4))  # raport

# ----------------------------
# 18.5) Confusion Matrix dla nowego progu
# ----------------------------

cm_thr = confusion_matrix(y_test, y_pred_thr)  # macierz pomyłek dla progu

plt.figure(figsize=(7, 5))  # rozmiar
sns.heatmap(  # heatmapa
    cm_thr,  # macierz
    annot=True,  # liczby
    fmt="d",  # int
    cmap="Greens",  # kolor
    xticklabels=["brak cukrzycy (0)", "cukrzyca (1)"],  # etykiety X
    yticklabels=["brak cukrzycy (0)", "cukrzyca (1)"]  # etykiety Y
)
plt.title("Macierz pomyłek (nowy threshold)")  # tytuł
plt.xlabel("Klasa przewidziana")  # oś X
plt.ylabel("Klasa rzeczywista")  # oś Y
plt.show()  # pokaż wykres
```

Literatura:

Python:

1. Lutz M., Python. Wprowadzenie, wyd. 6, Helion, 2025.
2. Matthes E., Python. Instrukcje dla programisty, wyd. III, Helion, Gliwice, 2023.
3. Lubanovic B., Introducing Python, wyd. 3, O'Reilly Media, 2025.
4. Lubanovic B., Python. Nowoczesne programowanie w prostych krokach, wyd. 2, Helion, 2020.
5. Sweigart A., Automate the Boring Stuff with Python, wyd. 3, No Starch Press, 2025.
6. Sarbicki G., Python. Kurs dla nauczycieli i studentów, wyd. 2, Helion, 2022.
7. Jaśniewski T., Python. Zbiór zadań z rozwiązaniami, Helion, 2024.

Analiza danych, uczenie maszynowe, uczenie głębokie:

1. Russell S., Norvig P., Sztuczna inteligencja. Nowe spojrzenie. Wydanie IV. Tom 1, Helion, 2023.
2. Russell S., Norvig P., Sztuczna inteligencja. Nowe spojrzenie. Wydanie IV. Tom 2, Helion, 2023.
3. Géron A., Uczenie maszynowe z użyciem Scikit-Learn, Keras i TensorFlow. Wydanie III, Helion, 2023.
4. Tabor J., Śmieja M., Struski Ł., Spurek P., Wołczyk M., Głębokie uczenie. Wprowadzenie, Helion, 2022.
5. Lakshmanan V., Robinson S., Munn M., Wzorce projektowe uczenia maszynowego. Rozwiązania typowych problemów dotyczących przygotowania danych, konstruowania modeli i MLOps, Helion, 2021.
6. Ameisen E., Uczenie maszynowe w aplikacjach. Projektowanie, budowa i wdrażanie, Helion, 2021.
7. Patel A. A., Praktyczne uczenie nienadzorowane przy użyciu języka Python, Promise, 2020.
8. Muraszkiewicz M., Nowak R. (red.), Sztuczna inteligencja dla inżynierów: metody ogólne, Oficyna Wydawnicza Politechniki Warszawskiej, 2022.
9. Muraszkiewicz M., Nowak R., Sztuczna inteligencja dla inżynierów. Istotne obszary i zastosowania, Oficyna Wydawnicza Politechniki Warszawskiej, 2023.

10. Moroney L., Sztuczna inteligencja i uczenie maszynowe dla programistów: praktyczny przewodnik po sztucznej inteligencji, Helion, 2021.
11. Howard J., Gugger S., Deep learning dla programistów: budowanie aplikacji AI za pomocą fastai i PyTorch, Helion, 2021.
12. Atienza R., Deep learning z TensorFlow 2 i Keras dla zaawansowanych. Sieci GAN i VAE, deep RL, uczenie nienadzorowane, wykrywanie i segmentacja obiektów i nie tylko. Wydanie II, Helion, 2022.
13. Tunstall L., von Werra L., Wolf T., Przetwarzanie języka naturalnego z wykorzystaniem transformerów. Budowanie aplikacji językowych za pomocą bibliotek Hugging Face, Helion, 2024.
14. Gallatin K., Albon C., Uczenie maszynowe w Pythonie. Receptury. Od przygotowania danych do deep learningu. Wydanie II, Helion, 2024.
15. Osowski S., Szmurło R., Matematyczne modele uczenia maszynowego w językach MATLAB i PYTHON, Oficyna Wydawnicza Politechniki Warszawskiej, 2024.
16. Grus J., Data science od podstaw. Analiza danych w Pythonie. Wydanie II, Helion, 2020.