Paweł Baranowski

Wirginia Doryń

PRZETWARZANIE DANYCH I UCZENIE MASZYNOWE W JĘZYKU PYTHON.
APLIKACJE W EKONOMII I ZARZĄDZANIU

Paweł Baranowski

Wirginia Doryń

PRZETWARZANIE DANYCH I UCZENIE MASZYNOWE W JĘZYKU PYTHON. APLIKACJE W EKONOMII I ZARZĄDZANIU

Instytut Badań Gospodarczych

Olsztyn 2020

Recenzenci:

dr hab. Michał Pietrzak, prof. UMK
dr Karol Korczak

Autorstwo rozdziałów:

Paweł Baranowski: rozdział 1., rozdział 2.8., załącznik do rozdziału 2., rozdział 3., załącznik do rozdziału 3., rozdziały 5.1–5.7

Wirginia Doryń: rozdziały 2.1.–2.7., rozdział 4., rozdział 5.8

Współautorstwo: rozdział 5.9

Skład, łamanie i projekt okładki (na podstawie Adobe Stock):

Ilona Pietryka

ISBN 978-83-65605-16-0

DOI: 10.24136/eep.mon.2020.1

Instytut Badań Gospodarczych
ul. ks. Roberta Bilitewskiego, nr 5, lok. 19
10-693 Olsztyn, Poland

biuro@badania-gospodarcze.pl
www.badania-gospodarcze.pl

Spis treści

Wprowadzenie

Książka, którą oddajemy w ręce Czytelników omawia nowoczesne rozwiązania w zakresie pobierania i przetwarzania danych. Przedstawiamy przykłady aplikacji w praktyce gospodarczej czy pracy naukowej, przy czym Czytelnik dostaje porcję wiedzy, a następnie niemal gotowe rozwiązania w języku Python. Dzięki elastycznej i intuicyjnej składni oraz bogatym zasobom, Python szczególnie nadaje się do pisania krótkich programów — np. skryptów do pozyskiwania danych i prostych analiz. Dlaczego w książce przedstawiamy implementację w Pythonie?

Python jest językiem łatwym w nauce, ma składnię zbliżoną do języka angielskiego, a przy tym jest bardzo zwięzły i wygodny w użyciu. Python jest interpretowany (czyli wykonywano na bieżąco bez konieczności kompilacji, czyli wygenerowania całego programu zapisanego z użyciem instrukcji niższego poziomu). Dzięki temu programy pisane w Pythonie są przenośne — działają niezależnie od systemu operacyjnego. Python jest też językiem zorientowanym obiektowo, co zapewnia elegancki i spójny kod programu i eliminuje wiele błędów wynikających z niedostosowania typów zmiennych (oczywiście kosztem większej „dyscypliny" na etapie projektowania programu; na szczęście dla nas ten problem w zasadzie nie dotyczy tematów poruszanych w tej książce).

W chwili pisania książki Python był jednym z najpopularniejszych języków programowania (w rankingu TIOBE za maj 2020 — trzeci z kolei, po C i Java) oraz najbardziej popularnym w zakresie przetwarzania danych (główni konkurenci w chwili pisania tej książki: R, Matlab/Octave, Julia, Ruby i Spark). Szerokie grono użytkowników Pythona sprawia, że dostępne są tysiące bibliotek (modułów). W dalszych rozdziałach zapoznamy Czytelnika z najbardziej popularnymi modułami służącymi do obróbki danych (*pandas*), tekstu (*NLTK*) oraz pobierania zawartości ze stron internetowych (*selenium*, *BeatifulSoup*). Dzięki oprogramowaniu nawet skomplikowanych zadań przetwarzania i obróbki danych, czyli zadań typowych dla kodu Python, zwykle nie pochłoną one długich

godzin. Wreszcie — Python jest dystrybuowany niekomercyjnie, na zasadach *open source*.

Struktura opracowania jest następująca. Rozdział 1 wprowadza w składnię Pythona i omawia podstawowe komendy, w tym instrukcje sterujące wykonaniem programu. W rozdziale 2 poznamy bliżej struktury danych w Pythonie. Oczywiście treści rozdziałów 1 i 2 nie są zupełnie rozłączne, od czasu do czasu programy z rozdziału 1 będą modyfikowały lub odczytywały dane zawarte w pamięci. Dlatego możliwe, że pewne szczegóły pominięte w rozdziale 1, zostaną doprecyzowane w rozdziale 2. Rozdział 3 przedstawia techniki automatycznego pobierania treści stron internetowych, przy pomocy technik *web scraping* lub serwisów API. W rozdziale 4 pogłębiamy wiedzę z zakresu przetwarzania danych w postaci tekstowej (napisów) oraz wprowadzamy zaawansowane techniki obróbki tekstu (*text mining*). Czytelnicy, którzy nie są zainteresowani tymi treściami, mogą pominąć rozdziały 3 i 4, a przejść od razu do technik uczenia maszynowego przedstawionych w rozdziale 5. Omawiając metody uczenia maszynowego, koncentrujemy się na zastosowaniach i składni Pythona. Z tego powodu ograniczyliśmy do minimum wzory i założenia matematyczne. Mamy nadzieję, że Czytelnicy zachęceni przez nas do użycia metod klasyfikacji i regresji, z czasem sięgną do podręczników z matematyki i metod ilościowych i zyskają nowe, głębsze spojrzenie na poznane metody.

Wszystkie przykłady przedstawione w książce zostały sprawdzone w Pythonie w wersjach 3.6 i 3.7, ale większość kodów powinno działać w dowolnej wersji od 3.5 wzwyż. W książce będziemy korzystać ze standardowej implementacji Pythona napisanej w języku C (Cpython). Zainteresowani Czytelnicy mogą zapoznać się różnicami w zakresie implementacji w książce Jaworskiego i Ziadé (2017, s. 30–33). Niezależnie od implementacji, dostępne są różnorodne dystrybucje Pythona. Z punktu widzenia niniejszej książki, warte zainteresowania są dystrybucje z dołączonymi standardowo bibliotekami do przetwarzania danych — np. Anaconda czy WinPython. Aktualna lista innych implementacji i dystrybucji znajduje się na stronie https://www.python.org/download/alternatives, a nieco szersze omówienie tych dystrybucji prezentuje Boschetti, Massaron (2017). W tej ostatniej pozycji można znaleźć też opis różnic pomiędzy Pythonem 3.x a Pythonem 2.x. Choć Python 2 jest już bardzo mało popularny, bywa wciąż używany w starszych projektach i sporadycznie zdarzają się jeszcze np. biblioteki, których nie „przepisano'' do Pythona w wersjach 3.x.

Autorzy dziękują recenzentom wydawniczym: Michałowi Pietrzakowi oraz Karolowi Korczakowi za cenne uwagi, które pozwoliły wyeliminować usterki i wzbogacić książkę o nowe wątki. Wcześniej mieliśmy przyjemność dyskutować fragmenty książki z Dariuszem Urbanem, Szymonem Wójcikiem i Adrianem Stępniakiem. Oczywiście za wszystkie pozostałe mankamenty książki odpowiedzialność ponoszą wyłącznie autorzy.

1. Wprowadzenie do programowania w Python

1.1 Ogólne zasady składni języka Python

Języki programowania różnią się, podobnie jak języki których używają ludzie. Przede wszystkim polecenia wydawane komputerowi za pomocą języka programowania muszą być wyrażone bardzo precyzyjnie. W przeciwieństwie do człowieka, komputer nie będzie pytał „czy na pewno chcesz wykonać taką komendę" (oczywiście twórca programu może przewidzieć taką możliwość i w określonym przypadku program będzie prosił o potwierdzenie; niemniej jednak komputer samodzielnie tego nie zrobi). Znajomość ogólnych zasad pisania kodu jest więc pierwszym krokiem nauki programowania w Python. Poniżej przedstawiamy pięć ogólnych zasad.

Zasada nr 1: Python rozróżnia duże i małe litery. Dotyczy to zarówno nazw komend (funkcji) jak i zmiennych. Przykładowo, komenda print() wyświetla liczbę (albo tekst), ale odwołanie się do komendy Print() zakończy się błędem.

```
>>> print(5)
5
>>> Print(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
```

Poniżej wprowadzimy do pamięci zmienną o nazwie 'liczba5' i przypiszemy jej wartość 5.

```
>>> liczba5 = 5
>>> print(liczba5)
5
>>> print(Liczba5)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Liczba5' is not defined
```

Zasada nr 2: Wcięcia za pomocą białych znaków (spacji, tabulatorów) na początku linii zmieniają sposób wykonania programu. Za pomocą wcięć („indentacji") oddzielane są bloki kodu, np. blok kodu wykonywany poza pętlą i wykonywany za każdym przebiegiem pętli (wewnątrz pętli). Ta własność jest dość unikalna dla Pythona. W innym językach zwyczajowo stosuje się wcięcia, ale mają one znaczenie z punktu widzenia czytelności oraz estetyki kodu, a nie zmieniają tego jak komputer wykonuje program — fragmenty kodu są oddzielane za pomocą innych znaków (np. '{' i '}'). Powrócimy do tego na końcu tego rozdziału, przy omawianiu komend sterujących wykonaniem programu (if-else oraz pętli).

Zasada nr 3: Komentarze oznaczamy za pomocą # (*hash*). Komentarze to fragmenty kodu, które są pomijane przez interpreter, ale są istotne dla ludzi którzy uczestniczą w tworzeniu programu. Umieszczamy tam wyjaśnienia dla innych programistów lub własne notatki, które pozwolą nam zrozumieć sens operacji wykonywanej w danym miejscu albo rolę jaką pełni przywoływana tu zmienna. Komentarz jednolinijkowy zaczyna się od znaku # i obowiązuje do końca linii.

Zasada nr 4: Python jest językiem w pełni zorientowanym obiektowo. Oznacza to, że wszystko w Python jest obiektem i podlega regułom programowania obiektowego. Dotyczy to nie tylko zmiennych, ale także funkcji. Na początku drogi z Python ma to niewielkie znaczenie, zwłaszcza że w tej książce nie omawiamy szczegółowych rozwiązań programowania obiektowego. Jednak zaznaczamy tę cechę, gdyż siłą rzeczy będziemy korzystać z „obiektowości" Python, a w dodatku nabierze to większego znaczenia na dalszych szczeblach poznawania języka.

Zasada nr 5: Sekwencje znaków, liczb czy innych obiektów (np. listy czy napisy, które szerzej będą opisane w rozdziale 2) są numerowane za pomocą indeksu, który zaczyna się od zera. Stosowana w Python reguła numerowania od zera (*zero-indexing*) jest zwykle używana w językach programowania ogólnego przeznaczenia (np. C++, Java, Ruby), natomiast w językach zaprojektowanych z myślą o obliczeniach (Matlab/Octave, R) używa się numerowanie od jeden (*one-indexing*).

Na zakończenie ogólnych informacji o składni Python, dodajmy że większość przykładów jest przeznaczona do wykonania komend w powłoce interaktywnej — czyli uruchamiamy interpreter Python a następnie wpisujemy lub wklejamy kod, linijka po linijce. Taki sposób uruchamiania kodu jest odpowiedni do nauki, a czasem do testowania fragmentów własnego kodu. Wraz z rozwojem umiejętności programowania, programy staną się bardziej rozbudowane lub zaprojektowane w celu wielokrotnego użycia. Wówczas będziemy zapisywać plik programu pliku (zwykle te pliki mają rozszerzenie .py) i uruchamiać je

wywołując interpreter Python wraz z nazwą pliku zawierającego kod źródłowy (np. wpisując w linii poleceń polecenie 'python.exe naszprogram.py').

1.2 Operatory przypisania i inne operatory liczbowe

Podstawowe operacje na zmiennych (obiektach) rozpoczniemy od operatora przypisania. Przypisanie modyfikuje zawartość pamięci komputera, umieszczając zadaną wartość pod adresem w pamięci przypisanym do danej zmiennej. Krótko mówiąc, przypisanie modyfikuje zawartość zmiennej. Przypisanie w Python realizowane jest poprzez *nazwazmiennej* = <wartość>, gdzie wartość jest wyrażona za pomocą stałej, np.:

```
>>> liczba10 = 10
lub pochodzi z innej zmiennej, np.:
>>> liczba10_drugiraz = liczba10
Oczywiście możliwe jest też przypisanie wyniku działania
na zmiennych lub stałych, np.
>>> liczba20 = liczba10 * 2
>>> liczba20 = liczba10 + liczba10_drugiraz
```

Operatora przypisania nie długich objaśnień. Chcielibyśmy tylko zwrócić uwagę na dwie kwestie. Po pierwsze, jak widać w powyższych przykładach, w Python nie deklaruje się zmiennych ani nie wymaga wskazania jakiego typu jest to zmienna. Przypisanie jednocześnie deklaruje nową zmienną, o ile ta nie istniała w przeszłości. Jeżeli zmienna o danej nazwie istniała, wówczas nadpisuje się nową wartość — tak jak w drugim przypisaniu wartości do 'liczba20' (i ewentualnie zmienia jej typ — Python sam o to zadba i nie musimy się tym przejmować). Po drugie, działanie operatora przypisania jest podobne dla wszystkich typów danych, można więc wprowadzać zmienne całkowitoliczbowe, zmiennoprzecinkowe, napisy i inne struktury danych, które zostaną omówione w rozdziale 2.

Krótkiego komentarza wymaga nazwa zmiennej. Python nie jest pod tym względem oryginalny — w nazwie zmiennej dopuszcza się litery, cyfry i znaki podkreślenia (_). Nazwa zmiennej nie może jednak zaczynać się od cyfry (dzięki temu unikamy niejednoznacznego lub mało czytelnego zapisu, który nie rozróżniałby nazw zmiennych od liczb). Nazwy zmiennych nie mogą pokrywać się z tzw. słowami kluczowymi (np. *True*, *False*, *if*, *else*; w dalszej części poznamy więcej takich słów kluczowych oraz ich zastosowanie). Generalnie, dobra praktyka zakłada, że nie nazwy zmiennych nie będą lakoniczne (np. 'x') i nie będą się pokrywać z określeniami, które Python wykorzystuje w innym kontekście[1]. Pisaliśmy już, że Python rozróżnia zmienne z dużej i małej litery, stąd 'Liczba' i 'liczba' będą oznaczały inne zmienne. Świadome wprowadzenie

[1] Na przykład: formalnie dopuszczalne jest przypisanie wartości do zmiennej 'list'. Jest to jednak o tyle niefortunne, że komendy list użyjemy do konwersji na listę.

dwu zmiennych o tak podobnych (dla człowieka) nazwach jest jednak proszeniem się o błąd w programie. Dobrze z kolei jeśli nazwy będą same z siebie wyjaśniać rolę danej zmiennej w programie. Oczywiście nie sposób wymienić wszystkich sytuacji niezalecanych, ale formalnie dopuszczalnych przez interpreter. Do kwestii dobrego stylu wrócimy jeszcze w rozdziale 1.8.

Do wartości liczbowych w Python możemy zastosować standardowe operatory dodawania (+), odejmowania (-), mnożenia (*) i dzielenia (/). Ilustruje to poniższy przykład:

```
>>> liczba1 = 4 - 2
>>> liczba2 = 2 * 3.14
>>> liczba3 = liczba2 / liczba1
>>> print(liczba1, liczba2, liczba3)
2 6.28 3.14
```

Poza tymi operatorami warto wymienić potęgowanie (**, czyli podwójna gwiazdka; uwaga! typowo w arkuszach kalkulacyjnych i innych językach programowania potęgowanie jest wykonywane za pomocą operatora ^, który w Python wyraża zupełnie inną operację[2]) oraz operator modulo (%), czyli reszta z dzielenia całkowitoliczbowego. Tym razem nie będziemy przypisywać wyniku do zmiennych, ale wyświetlimy od razu wyniki za pomocą funkcji print(). Dodatkowo, w poniższym kodzie umieściliśmy komentarze.

```
>>> print(2 ** 8)  # wyświetl wynik: 2 do potęgi 8
256
>>> print(15 % 8)  # wyświetl wynik: reszta z dzielenia 15
przez 8
7
```

Często chcemy powiększyć lub pomniejszyć aktualną wartość zmiennej, np. o dwa. Możliwe jest oczywiście skorzystanie z przypisania *a = a + 2* oraz *a = a - 2*. Możemy takie działania zapisać krócej za pomocą operatorów *inplace*: przypisania-dodawania (+=) i przypisania-odejmowania (-=)[3].

```
>>> a = 5.5
>>> a += 2   # wynik ten sam co w przypadku a = a + 2
>>> print(a)
7.5
>>> a -= 2   # wynik ten sam co w przypadku a = a - 2
```

[2] Gwoli ścisłości: alteratywę wykluczającą (XOR) wykonaną na reprezentacji binarnej obu argumentów. Dlatego wynikiem operacji 3 ** 3 będzie liczba 27, podczas gdy 3 ^ 3 zwróci liczbę 0.

[3] Używając operatorów += czy -= zwiększamy także szybkość działania programu, choć w przypadku operacji liczbowych w Python różnica jest nieznaczna (co najwyżej kilka procent). Dużo większe skrócenie czasu obliczeń zyskujemy używając operatorów *inplace* do obliczeń wektorowych w pakiecie *numpy* (zob. Gorelick, Ozsvald, 2020, s. 126).

```
>>> print(a)
5.5
```

Analogiczne działanie mają inne operatory *inplace*: przypisania-mnożenia (*=), przypisania-dzielenia (/=), przypisania-potęgowania (**=) i rzadko stosowane przypisania-modulo (%=). Przykłady pozostawiamy Czytelnikowi.

Tab. 1.1 przedstawia podsumowanie operatorów (wybranych), dodatkowo przedstawiając je według kolejności wykonywania działań (na początku operatory o najwyższym priorytecie). Operacje o tym samym priorytecie są wykonywane od lewej do prawej.

Tab. 1.1 Wybrane operatory, według kolejności wykonywania działań

Operatory	Znaczenie
-a	Minus (operator jednostronny, zmienia znak na przeciwny)
a ** b	Potęgowanie (‘a’ do potęgi ‘b’)
x * y, x / y, x // y, x % y	Mnożenie, dzielenie, dzielenie całkowitoliczbowe, reszta z dzielenia
x + y, x - y	Dodawanie, odejmowanie
x < y, x <= y, x > y, x >= y	Operatory relacyjne: mniejszy, mniejszy równy, większy, większy równy
x == y, x != y	Operatory relacyjne: równy, różny
a in zbior_wartosci, a not in zbior_wartosci	Czy ‘a’ znajduje (zawiera) się w ‘zbior_wartosci’ (operator ten będzie szerzej omówiony w rozdziale 2), czy ‘a’ nie znajduje się w ‘zbior_wartosci’
not a	Negacja (operator logicznego zaprzeczenia)
x and y	Oraz (operator logicznego iloczynu)
x or y	Lub (operator logicznej sumy)

Najwyższy priorytet ma jednostronny operator minus (negacja), kolejne trzy poziomy odpowiadają działaniom arytmetycznym (a ich kolejność wykonywana jest zgodna z kolejnością działań w matematyce). Po operacjach arytmetycznych wykonywane są wyniki operatorów relacyjnych (sprawdzenie warunku), a następnie operatory logiczne. Warto zapamiętać, że kolejność działań logicznych jest podobna jak działań arytmetycznych (najpierw negacja, która odpowiada jednostronnemu minus, następnie iloczyn logiczny a na końcu suma logiczna).

Na koniec dodajmy, że działanie operatorów może być różnie zaimplementowane w zależności od typu. Wybiegając nieco w przyszłość, operator dodawania złączy dwie listy w jedną dłuższą (zob. rozdz. 2.3), choć równie dobrze można by się spodziewać, że zwróci wyniki dodawania ich kolejnych elementów (zwłaszcza w przypadku, gdy obie listy mają jednakową długość). Ale już w bibliotece *pandas* dodając dwa obiekty typu *Series* w wyniku otrzymamy właśnie dodawanie kolejnych elementów szeregu (zob. rozdz. 2.8). Ponadto

dla niektórych typów danych nie wszystkie opisane wyżej operatory są zdefiniowane (np. odejmowanie czy dzielenie list).

1.3 Operatory porównania, relacji i operatory logiczne

W trakcie wykonywania programu często porównujemy dwie wartości i od wyniku tego porównania uzależniamy wykonanie fragmentu kodu (np. instrukcja warunkowa 'if', rozdz. 1.5).

Podstawowym tego typu operatorem jest porównanie (==, czyli podwójna równość, uwaga: nie mylić z operatorem przypisania =). Wynikiem tej operacji jest wartość logiczna (bool) przyjmująca wartość prawda albo fałsz ('True' albo 'False').

```
>>> print(2 == 5)
False
>>> a = 5
>>> print(a == 5)
True
```

Przeciwieństwem operatora relacji równości jest operator nie-równy (!=). Podobnie możemy sprawdzać inne relacje pomiędzy wartościami numerycznymi, np. większy (>), większy lub równy (>= ALE: uwaga na kolejność, bo => nie zadziała), mniejszy (<), mniejszy lub równy (<=).

Oczywiście często potrzebujemy sprawdzić złożone warunki. Na przykład, chcemy sprawdzić czy liczba jest większa od pięciu a jednocześnie parzysta (warunek ten można zapisać następująco: czy modulo z dzielenia przez dwa jest równe zero). Możemy to uzyskać korzystając z operatora logicznego „oraz'' (and). Operator 'and' zwróci wartość True tylko gdy oba składniki będą prawdziwe.

```
>>> liczba = 25
>>> print(liczba > 5)
True
>>> print(liczba % 2 == 0)  # czy ‘liczba' jest parzysta
False
>>> print(liczba > 5 and liczba % 2 == 0)
False
```

Inne bardzo użyteczne operatory logiczne to zaprzeczenie (not; operator ten umieszczamy przed wynikiem innej operacji logicznej) oraz operator „lub'' (or). Krótko mówiąc, not zmienia True na False i odwrotnie, a wynik operacji 'or' daje True, jeśli przynajmniej jeden ze składników jest prawdziwy.

Dla tych, którzy nie znają lub nie pamiętają działania tych operatorów, przedstawiamy krótką ilustrację.

```
>>> print(20 > 50)
False
>>> print(not 20 > 50)
True
>>> print(not False, not True)
True False
>>> print(False or False, True or False, False or True,
True or True)
False True True True
```

Podobnie jak w przypadku operacji liczbowych, możemy przypisać wynik porównania czy operacji logicznej do zmiennej. W efekcie otrzymamy zmienną typu logicznego (bool). Typów zmiennych jest oczywiście więcej, a szczegóły poznamy w rozdziale 2.

```
>>> zmienna_boolean = (12 > 10)
>>> print(not zmienna_boolean)
False
>>> print(False or zmienna_boolean)
False
>>> druga_boolean = not (10 == 5)  # to samo co wynik ope-
racji 10 != 5
```

1.4 Import modułów (bibliotek) i wywołanie funkcji

Im bardziej popularny język programowania, tym więcej posiada gotowych narzędzi, które wykonują różne pożyteczne czynności (np. obliczają wynik złożonych obliczeń, rysują wykres, wysyłają maile, zapisują dane do arkusza Excel itp.). Narzędzie, które służy do wykonywania jednego typu operacji to funkcja. W dotychczasowych ćwiczeniach korzystaliśmy z wbudowanej[4] funkcji print(). Moduł (w innych językach określany jako biblioteka lub pakiet) to nic innego jak zbiór funkcji, zwykle zebranych „tematycznie".

Funkcję wywołujemy podając jej nazwę i argumenty w nawiasie. Poniżej pokażemy kilka prostych przykładów z użyciem innej funkcji — round(). Funkcja jest opisana w dokumentacji następująco: `round(`*`number`*`[, `*`ndigits`*`])` i oblicza zaokrąglenie liczby '*number*' do podanej liczby miejsc dziesiętnych '*ndigits*'. Drugi argument jest opcjonalny, jeśli nie podamy liczby miejsc dziesiętnych, funkcja obliczy zaokrąglenie do pełnych liczb całkowitych.

```
>>> round(1.47)
1
>>> round(1.47, 1)
1.5
>>> round()
```

[4] Funkcja wbudowana to taka, która jest standardowo dostępna w języku programowania i nie wymaga korzystania z dodatkowych modułów.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Required argument 'number' (pos 1) not found
>>> round("Alamakota")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: type str doesn't define __round__ method
```

Dwa pierwsze wywołania funkcji round() nie powinny budzić wątpliwości — funkcja zwróciła prawidłowe wartości. Krótkiego komentarza wymaga natomiast trzecia i czwarta komenda. Trzecia komenda zakończyła się błędem, gdyż nie podano argumentu obowiązkowego (i Python nie wiedział jaką liczbę chcemy zaokrąglić). Czwarta komenda również zakończyła się błędem, mimo że podano wymagany argument. Tym razem błąd wynika stąd, że funkcja round() „przyjmuje" wyłącznie zmienne liczbowe, a podano ciąg liter (napis, string).

W przypadku funkcji przyjmującej więcej niż jeden argument możemy podawać argumenty dwojako. Pierwszy sposób to argumenty według ich kolejności („pozycyjnie"). Tak wywoływaliśmy funkcję round() w przykładzie powyżej (`round(1.47, 1)`). Drugi sposób to podanie nazw argumentów — wówczas funkcja rozpozna argumenty według nazw i zwróci tę samą wartość niezależnie od kolejności podania argumentów.

```
>>> round(number=1.47, ndigits=1)
>>> 1.5
>>> round(ndigits=1, number=1.47)
>>> 1.5
```

Dzięki czemu nie musimy pamiętać kolejności argumentów a także łatwiej zrozumiemy intencje twórcy programu. Taki sposób jest zalecany, gdy podajemy trzy lub więcej argumenty.

W jaki sposób korzystać z modułów? Do tego służy polecenie import. Możemy zaimportować cały moduł albo tylko wybraną funkcję[5]. W pierwszym przypadku wywołanie funkcji będzie w postaci mojmodul.funkcja(), a w drugim tylko funkcja(). Oba sposoby są równie często stosowane, a różnice między nimi przedstawimy przez analogię do zdejmowania książki z półki — w pierwszym przypadku wypożyczamy całą książkę, dlatego odwołując się musimy podać nazwę książki i numer strony. W drugim robimy kserokopię tylko jednej strony i nie musimy już podawać nazwy książki, ale trzeba podać która strona interesuje nas z danej książki.

Poniższe przykłady ilustrują użycie funkcji ceil() („sufit", zaokrąglenie w górę) i floor() („podłoga", zaokrąglenie w dół) z modułu *math* (Czytelni-

[5] Możemy także importować stałe, na przykład stałą *pi* = 3.141592... (import math.pi).

kowi pozostawiamy sprawdzenie w dokumentacji pozostałych funkcji z tego modułu).

```
>>> import math
>>> math.ceil(1.41)
2
>>> math.floor(1.41)
1
```

Jeśli nie chcemy za każdym razem odwoływać się do modułu *math*, a bezpośrednio do nazw funkcji ceil() i floor(), wówczas możemy użyć drugiego sposobu:

```
>>> from math import ceil, floor
>>> ceil(1.41)
2
>>> floor(1.41)
1
```

Co istotne, w Python polecenie import nie musi być umieszczone na początku programu (choć jest to zwyczajowo przyjęte). Ważne jedynie, żeby import wykonać przed pierwszym wywołaniem funkcji.

Python 3.7 posiada około 100 wbudowanych modułów[6]. Tab. 1.2 przedstawia krótką listę modułów „ogólnego przeznaczenia", najbardziej istotnych z punktu widzenia książki.

Tab. 1.2 Wybrane moduły wbudowane Python

Biblioteka	Zastosowanie
os	Obsługa plików (odczyt, zapis, zmiana katalogu itd.)
datetime	Obsługa dat (z dokładnością do mikrosekund), operacje matematyczne na datach
re	Obróbka tekstu za pomocą tzw. wyrażeń regularnych (zob. rozdz. 4.2)
random	Liczby pseudolosowe
urllib	Pobieranie zawartości stron HTML
math	Funkcje matematyczne (np. pierwiastek kwadratowy, sinus, logarytm)
copy	Kopia obiektu (zob. rozdz. 1.7)

Ostatnim zagadnieniem jest instalacja modułów, które nie są domyślnie zawarte w danej wersji (implementacji) interpretera. Do instalacji i zarządzania bibliotekami służy polecenie 'pip', standardowo dostarczane razem z Python w wersji 3.4 i wyżej (UWAGA: polecenie to jest wywoływane z linii poleceń systemu operacyjnego, a nie w interpreterze Python!).

Do instalacji modułu trzeba wpisać w linii poleceń komendę *pip install 'nazwabiblioteki'*, natomiast pip list wyświetli listę zainstalowanych modułów

[6] https://docs.python.org/3/library.

i ich wersji (moduły są aktualizowane, dlatego wersja może mieć znaczenie). Na wszelki wypadek dodamy, że do instalacji nowych modułów niezbędne jest działające połączenie z internetem. Doskonałą okazją do sprawdzenia przedstawionej wyżej komendy będzie instalacja modułu *pandas* (moduł ten będzie szerzej omówiony i używany w dalszej części książki).

1.5 Instrukcje warunkowe: if, if-else, if-elif-else

Podstawową instrukcją umożliwiającą sterowanie wykonaniem programu jest instrukcja warunkowa ('if'). Instrukcja warunkowa umożliwia uzależnienie wykonania fragmentu kodu od tego, czy warunek logiczny jest spełniony (czyli przyjmuje wartość True). W praktyce większość naszych programów będzie sprawdzało warunek i uzależni swoje działania od wyniku tego sprawdzenia. Przykładowo w grze w „statki" możemy sprawdzić czy pole wskazane przez gracza atakującego na planszy drugiego gracza jest zajęte — jeśli tak, to oznaczamy strzał jako trafiony itd.

W programach przetwarzających dane często instrukcja warunkowa służy do weryfikacji poprawności wprowadzonych danych. Na przykład, w programie, która ma podsumować przelewy do określonego kontrahenta, możemy najpierw upewnić się, że podany kontrahent otrzymał przynajmniej jeden przelew. Natomiast jeśli program przetwarza arkusz Excela, możemy najpierw sprawdzić czy plik o podanej nazwie istnieje (i dopiero jeśli istnieje — wczytujemy go, przetwarzamy i wyświetlamy wynik).

Składnia instrukcji (słowa kluczowego) if:

```
if testowany_warunek:
_ _ _ _instrukcje wykonywane jeśli testowany_warunek jest
prawdziwy
_ _ _ _dalsze instrukcje wykonywane jeśli testowany_warunek
jest prawdziwy
instrukcje wykonywane niezależnie od testowanego warunku
```

(UWAGA: znaki _ _ _ _ podkreślają, że chodzi o wcięcie w kodzie, ale są to spacje — typowo cztery spacje albo tabulator oznaczają jeden poziom wcięcia; dalej, z wyjątkiem prezentacji składni instrukcji if oraz for, nie będziemy stosować tego podkreślenia, a jedynie spacje; w kodzie z wyróżnionymi poziomami kodu pojawiają się trzy kropki na początku wersu — po tym jak interpreter spodziewa się kolejnego poziomu kodu, wyświetla '...', ale podobnie jak znaku zachęty '>>>" nie należy wpisywać trzech kropek).

Przypominamy Czytelnikowi, że wcięcie w kodzie wyróżnia określony blok kodu. W tym przypadku jest będą to polecenia, które zostaną wykonane jeśli warunek jest prawdziwy. W Python wcięcia są wymagane m.in. po instrukcjach warunkowych, pętlach i definicjach funkcji. W każdym z tych przypadków

pierwsza linia kończy się dwukropkiem, tak więc warto zapamiętać, że po dwukropku wymagane jest wcięcie kodu.

Równie prosta jak składnia jest zasada działania *if*, trzeba tylko pamiętać o wcięciach, które będą wyróżniały blok kodu wykonywany w razie spełnienia sprawdzanego warunku. Przykład podany niżej sprawdza czy 7 jest liczbą dodatnią, a następnie czy jest liczbą ujemną (a jeśli tak to wyświetla odpowiedni opis).

```
>>> if 7 > 0:
... print("7 to liczba dodatnia")
...
7 to liczba dodatnia
>>> if 7 <= 0:
... print("7 nie jest liczbą dodatnią")
...
```

Oczywiście po słowie kluczowym if możemy umieścić nie tylko prosty warunek, ale także warunki złożone (określona z użyciem opisanych operatorów logicznych, zob. przykłady w rozdz. 1.7). Jeśli zachowaliśmy wynik pewnej operacji logicznej w zmiennej (typu bool), wówczas możemy napisać: *if zmienna*: , gdzie `zmienna` będzie zmienną typu logicznego[7].

Rozważmy dalej nasz przykład z liczbą dodatnią i niedodatnią. Widzimy, że jeśli pierwszy warunek nie jest spełniony, wówczas drugi zawsze spełniony (i na odwrót). W takim przypadku możemy wprowadzić rozgałęzienie w kodzie, tzn. napisz „7 to liczba dodatnia'' jeżeli pierwszy warunek jest spełniony, a jeśli warunek ten nie jest spełniony — napisz „7 nie jest liczbą dodatnią''. W ten sposób działa instrukcja if-else. Poniżej przedstawiamy przykład z dwukrotnym użyciem instrukcji 'if' przepisany na krótszą i szybciej wykonywaną wersję 'if-else' (obecnie sprawdzamy warunek tylko raz). Nie trzeba zapewne dodawać, że tutaj także istotne są wcięcia w kodzie.

```
>>> if 7 > 0:
...     print("7 to liczba dodatnia")
... else:
...     print("7 nie jest liczbą dodatnią")
...
7 to liczba dodatnia
```

Oczywiście działanie obu fragmentów kodu jest identyczne. Dlaczego więc przedstawiamy instrukcję 'if-else', skoro może być zastąpiona dwukrotnym

[7] Ściślej rzecz biorąc, 'zmienna' może być zmienną innego typu. W takim przypadku Python spróbuje wykonać operację „rzutowania'' (zmiany typu zmiennej) na typ logiczny. Operacje te opisujemy w rozdziale 2. Zwykle rzutowanie na typ logiczny przebiega następująco: wartość niezerowa jest zamieniana na True (w przypadku typów liczbowych) albo wartość niepusta jest zamieniana na True (w przypadku listy, napisów i innych typów składających się się z większej liczby elementów).

'if'? Kod napisany z użyciem 'if-else' jest nie tylko krótszy, ale jednoznacznie wyrażamy, że drugi blok jest wykonywany w przypadku gdy wyrażenie przy 'if' ma wartość False. W ten sposób nasz program będzie mniej podatny na błędy programistyczne. Przykład z życia to sytuacja, w której twórca programu zmienia warunek przy pierwszym 'if' (np. teraz interesuje nas, czy podana wartość jest większa od 9) i zapomina zmienić drugi warunek. Jak łatwo sprawdzić taki kod może zadziałać w sposób nieprawidłowy. Czytelnikowi pozostawiamy analizę poniższego fragmentu (oraz samodzielne przepisanie go na kod z użyciem 'if-else').

```
>>> if 7 > 9:
...     print("7 to liczba większa od 9")
...
>>> if 7 <= 0:
...     print("7 nie jest większa od 9")
...
```

Co w przypadku gdy chcemy testować rozgałęzienie na kilka podgałęzi (opcji)? Przykładowo, chcemy określić czy liczba jest „co najmniej dwucyfrowa", „jednocyfrowa i dodatnia" czy „niedodatnia" i w zależności od tego wyświetlić komunikat (a w przykładach bliższych rzeczywistości: wykonać jakąś operację na zmiennych).

Pierwszy sposób to sprawdzamy czy liczba jest większa od 10, jeśli nie jest to sprawdzimy czy jest większa od zera, a jeśli i to nie jest spełnione. Możemy to osiągnąć „zagnieżdżając" w sobie dwa warunki if-else, tak jak w poniższym przykładzie (zwróćmy uwagę, że w przykładzie występują także podwójne wcięcia oznaczające blok kodu leżący wewnątrz innego bloku kodu).

```
>>> if liczba > 10:
...     print("Liczba co najmniej dwucyfrowa")
... else:
...     if liczba > 0:
...         print("Liczba jednocyfrowa i dodatnia")
...     else:
...         print("Liczba ujemna")
...
Liczba jednocyfrowa i dodatnia
```

Większą czytelność osiągniemy używając instrukcji if-elif-else (jedna z dobrych praktyk programowania w Python mówi, że operacje o mniejszym stopniu zagnieżdżenia są lepsze[8]; *Flat is better than nested*).

```
>>> if liczba > 10:
...     print("Liczba co najmniej dwucyfrowa")
```

[8] Pełną listę postulatów programowania w „duchu" Pythona (co określamy jako *Pythonic*) możemy wyświetlić wpisując w interpreter Pythona komendę 'import this'. Dobre praktyki pisania kodu zwięźle omawiamy w rozdz. 1.8.

```
... elif liczba > 0:
...     print("Liczba jednocyfrowa i dodatnia")
... else:
...     print("Liczba ujemna")
...
Liczba jednocyfrowa i dodatnia
```

W instrukcjach if-elif-else dużą rolę ma kolejność, a jej niezachowanie może prowadzić do zachowania programu niezgodnie z naszymi intencjami. Najpierw sprawdzany jest warunek if, jeśli nie jest on spełniony wówczas kolejno sprawdza się warunki elif. (Dodajmy w tym miejscu, że choć powyższym przykładzie znajduje się jeden warunek elif, to w warunków elif może być więcej) Dopiero jeśli żaden z warunków podanych po elif nie jest spełniony to wykonuje się blok else.

Ciekawostka: zamiast warunku zapisanego przy pomocy wyrażenia możemy także umieścić zmienną logiczną (boolowską), zmienne liczbowe (wartości niezerowa będą traktowane jako True), a nawet typy złożone (np. listy, krotki, itp. — omówione w rozdz. 2). Ogólna zasada jest taka, że dowolna niepusta wartość takiej zmiennej jest traktowana jak True.

1.6 Pętla for, sekwencje range()

Pętla jest kolejną instrukcją sterującą wykonaniem programu. Dzięki niej możemy wykonać pewien fragment kodu wielokrotnie, dla różnych wartości (oraz różnych elementów złożonych typów danych, o czym przekonamy się w rozdziale 2).

Pętla jest bardzo przydatną konstrukcją i umożliwia wykonanie tego samego zestawu instrukcji dla wielu obiektów. Przykładowo, mając zestaw danych finansowych dla kilkuset firm, możemy dla każdej z nich wykonać obliczenie wskaźników rentowności i płynności[9]. Podobnie gdy przeliczamy kwotę faktury na złote, chcielibyśmy wskazać programowi: przeliczyć w ten sposób wszystkie faktury, którymi dysponuję (przykład taki omówimy na końcu rozdz. 2). W zbiorze stu tysięcy maili możemy wyświetlić te e-maile, które zawierają określony wzorzec tekstu (podobne zagadnienia będą omówione w rozdz. 4.2). Bez wielkiej przesady można powiedzieć, że pętle stosujemy niemal w każdym programie przeznaczonym do pobierania czy analizy danych.

Składnia pętli for jest następująca:

```
for x in zakres_wartosci:
_ _ _ _instrukcje(x)   # wciecia oznaczaja blok kodu
```

[9] Zrozumienie tego przykładu nie wymaga znajomości wymienionych wskaźników. Czytelników zainteresowanych tą tematyką odsyłamy np. do książki Gabrusewicz (2019).

Powyższa pętla wykonuje instrukcje (których wykonanie w jakiś sposób powinno zależeć od wartości zmiennej x) dla wszystkich wartości podanych poprzez `zakres_wartosci`. Już teraz zwrócimy uwagę, że sposób podania tych wartości jest bardzo różny — może być sekwencja albo zmienna składająca się z większej liczby elementów (np. lista czy krotka).

W prostych pętlach, w których wykonujemy polecenia czy obliczenia dla kolejnych liczb całkowitych używamy `range([start,] stop, [step])`, gdzie dwa argumenty — start i step są opcjonalne (wartości domyślne odpowiednio 0 i 1, oznaczają że pierwszą liczbą będzie zero, a kolejne wartości będą różniły się o jeden). W ten sposób możemy wyświetlić np. kwadraty kolejnych liczb od 0 do 5.

```
>>> for i in range(6):
...     print(i**2)
...
0
1
4
9
16
25
```

Uważni Czytelnicy zapewne zwrócili uwagę, że w pierwszej linii podano range(6), podczas gdy program zakończył wykonanie na kwadracie liczby 5. Jest to zachowanie zgodne z konwencją, którą można zapamiętać następująco. Sekwencja range(6) zatrzyma się w razie napotkania wartości 6 i już nie wykona bloku kodu wewnątrz pętli. Równie dobrze można zapamiętać, że range(6) wyświetli 6 kolejnych liczb całkowitych, począwszy od zera (czyli: 0, 1, 2, 3, 4, 5).

Do pętli wrócimy jeszcze w rozdziale 2, ale już teraz Czytelnikowi zalecamy wykonanie pętli wyświetlającej daną liczbę pomnożoną przez 10 np. z sekwencjami: range(1, 9, 2) oraz range(0, 100, 10).

1.7 Bardziej złożone zagadnienia

Przegląd nieco bardziej złożonych (czasami również mniej znanych) operacji rozpoczniemy od łączenia dwu warunków logicznych. Wyobraźmy sobie, że chcemy sprawdzać ceny wybranych produktów w sklepie, a konkretnie czy cena jest „okrągła'' (wynik True/False może być użyty np. w instrukcji warunkowej). „Okrągła'' definiujemy następująco: jest większa od 10 i jednocześnie podzielna przez 5. Ale w przypadku cen mniejszych lub równych 10 wystarczy nam, jeśli cena będzie podzielna przez 2. Wynik takiej operacji sprawdzimy poprzez łączenie operacji logicznych and/or. Czyli: (sprawdź czy cena jest większa niż 10 oraz czy cena jest podzielna przez 5) lub (sprawdź czy cena jest mniej-

sza równa 10 oraz czy cena jest podzielna przez 2). W kodzie Python będzie to wyglądać następująco:

```
>>> cena = 15
>>> print(cena > 10 and cena % 5 == 0 or cena <= 10 and
cena % 2 == 0)
True
>>> cena = 9
>>> print(cena > 10 and cena % 5 == 0 or cena <= 10 and
cena % 2 == 0)
False
```

Oczywiście 15 jest ceną „okrągłą", ale 9 nie. W kodzie przedstawionym poniżej nie ma nawiasów przy operacjach and. Nie są one potrzebne, gdyż zgodnie z kolejnością działań (Tab. 1.1) operacja and jest wykonywana przed operacją lub. Tak więc np. dla 15 Python najpierw policzy wynik operacji: cena > 10 and cena % 5 (True) i operacji: cena <= 10 and cena % 2 == 0 (False). Następnie policzy wynik pierwszego warunku (który był True) lub drugi warunek (który był False) (finalnie otrzymując wynik działania True or False, który jest równy True).

Python akceptuje wielokrotne przypisania, tzn. w jednej linii kodu możemy przypisać kilka wartości do kilku zmiennych. Poniższy przykład oferuje dwa sposoby takiego przypisania (przy czym pierwszy jest bardziej użyteczny, w drugim musimy przypisać tę samą wartość do wszystkich zmiennych).

```
liczba1, liczba2, liczba3 = 1, 2, 3
liczba1 = liczba1_drugiraz = 1
```

Niekiedy chcemy zatrzymać działanie pętli for. Na przykład gdyby nasz program za pomocą pętli miał wybierać produkty, które kupimy w sklepie, pętla powinna zatrzymać swoje działanie gdy przekroczymy budżet. Służy do tego instrukcja 'break'. Za każdym razem gdy wywołamy wewnątrz pętli 'break', Python „wyjdzie" z pętli. Spójrzmy na przykład, który wyświetla kwadraty liczb całkowitych.

```
>>> for i in range(10):
...     print(i**2)
...     if i >= 5:
...         break
...
0
1
4
9
16
25
```

Jak widać, pętla operuje na liczbach całkowitych od 0 do 9 włącznie — range(10). Jednak w przypadku, gdy zmienna 'i' osiągnie wartość 5, pętla kończy swoje działanie, wcześniej wyświetlając kwadrat liczby 5. Przykład ten ma na celu tylko ilustrację działania instrukcji break. Z punktu widzenia dobrego zasad programowania (rozwiązania prostsze są lepsze — *Simple is better than complex* oraz zachowuj czytelność kodu — *Readability counts*) nie powinno tak się robić: efekt osiągamy za pomocą zbyt skomplikowanego i mniej czytelnego kodu, w rzeczywistości użylibyśmy range(6) tak jak w przykładzie z rozdz. 1.6).

Już na początku podkreślaliśmy, że wcięcia w Python oddzielają blok kodu (np. w operacjach warunkowych czy pętlach). Istnieją jednak wyjątki od tej zasady. Najważniejszym takim wyjątkiem są komendy (wyrażenia), które nie mieszczą się w jednej linii kodu. Gdy wyrażenie jest dłuższe niż typowa liczba znaków w jednej linii (zwykle od 75 do 80), możemy je wprowadzić „łamiąc linię", np. po otwarciu nawiasów (zwykłych, kwadratowych i klamrowych). Kolejne wiersze zaczynamy od wcięcia, czym sygnalizujemy że wiersze te stanowią kontynuację. Przykładowo, wprowadzając listę możemy skorzystać z tego zapisu:

```
>>> dlugalista = [12500, 12700, 12900,
...               13300, 13500, 13400]
```

Tak jak w podanym wyżej przykładzie, możliwe jest stosowanie dwu lub więcej wcięć (pomaga to zachować czytelność). Musimy jedynie pamiętać, że gdy jest więcej kontynuowanych linii, takie podwójne (wielokrotne) wcięcia muszą być stosowane we wszystkich liniach.

Łamanie linii będzie bardzo przydatne w przypadku wywołania funkcji zawierającej wiele argumentów, na przykład:

```
>>> najwieksza_liczba = max(125, 240, 256,
...                         296, -100, 200, 100, 153)
```

Na końcu drobna uwaga: liczby niecałkowite są reprezentowane w pamięci tylko z określoną dokładnością (zob. liczby zmiennoprzecinkowe, rozdz. 2; szersze omówienie precyzji rozwiązań układów równań i innych zagadnień algebry liniowej opartych na liczbach zmiennnoprzecinkowych — Meyer, 2000, s. 21 i nast.). Dlatego niektóre operacje zmiennoprzecinkowe mogą dać wynik przybliżony. Takim nieoczekiwanym wynikiem jest:

```
>>> print(0.3 + 0.3 + 0.3 == 0.9)
False
```

Oczywiście powyższa suma matematycznie jest równa 0.9, ale zarówno liczba 0.9 jak i liczby 0.3 są przechowywane z przybliżeniem, bardzo dokładnym ale jednak przybliżeniem[10]. Wynik ten jest matematycznie niepoprawny,

[10] Przybliżenie to sięga około 16–17 miejsca po przecinku.

ale nie popełniliśmy błędu w kodzie! Problemem nie jest to, że Python błędnie sumuje. Python dobrze sumuje, ale reprezentacja zmiennoprzecinkowa przechowuje ułamki z pewnym przybliżeniem. Na szczęście, w większości przypadków operacje zmiennoprzecinkowe są wykonywane wystarczająco dokładnie.

1.8 Kilka słów o stylu programowania i konwencjach

W rozdziale przywołaliśmy już dobre praktyki programowania. Styl pisania kodu programu jest w dużej mierze sprawą indywidualną, jednak zastosowanie się do kilku wskazówek sprawi, że kod będzie bardziej zgodny z zamierzeniami twórcy (twórców) języka Python (co określamy przymiotnikiem *Pythonic*). W rezultacie nasz kod będzie bardziej efektywny i mniej podatny na błędy. Początkujący programiści zwykle koncentrują się na tym, żeby ich kod działał (brak błędów składniowych i semantycznych) a w drugiej kolejności żeby poprawnie wykonywał zamierzone działanie (brak błędów logicznych). Natomiast styl pisania kodu jest zwykle dość przypadkowy. Znaczenie dobrego, spójnego stylu kodowania jest o tyle większe, że większość Czytelników tej książki nie pracuje (jeszcze?) w zawodzie programisty. W takim przypadku nie mają oni silnie narzuconych (przez przełożonych, kolegów z zespołu czy „korporacyjne" zasady) dobrych wzorców pisania kodu.

Dwa podstawowe źródła wskazujące na poprawny styl kodu znajdują się w dokumentacji języka, mianowicie: PEP-20 (lista tych ogólnych wskazówek jest również dostępna za pomocą komendy 'import this') i PEP-8. Omówimy je krótko, zachęcając Czytelnika do samodzielnej lektury, równolegle z rozwojem umiejętności programowania w Python (obszerne omówienie w języku polskim, np. Slatkin, 2015).

PEP-20 zawiera listą 19 zaleceń napisanych bardzo literackim językiem. Stanowią one bardziej przesłanie i opis filozofii języka niż sztywne reguły. Główne wskazówki płynące z tych zaleceń można podsumować następująco:

– pisz zwięźle, korzystaj z prostych rozwiązań i nie utrudniaj kodu (Python często oferuje niestandardowe rozwiązania, które owocują zwięzłym kodem, staraj się wykorzystywać te możliwości),
– unikaj nadmiernego zagnieżdżenia kodu (kolejne poziomy utrudniają zrozumienie kodu przez człowieka i zwiększają ryzyko błędu, szczególnie że Python jest wrażliwy na poziom indentacji),
– zachowuj czytelność kodu i unikaj wieloznaczności.

Od siebie dodamy także wskazówkę dotyczącą nazw (zmiennych, funkcji i innych obiektów). Początkujący programiści stosują krótkie nazwy (np. 'a', 'b' czy 'x1', 'x2'). Nazwy te niewiele mówią, w rezultacie po krótkiej przerwie w pisaniu nawet sam autor programu ma problemy ze zrozumieniem przypomnieniem sobie do czego służy dana zmienna czy funkcja. Lepszym rozwiązaniem

jest więc wprowadzać nazwy, które same wyjaśniają znaczenie obiektu (np. 'lista_plikow', 'suma_wydatkow').

Z kolei dokument PEP-8 precyzuje konwencję zapisu poleceń i zmiennych w języku Python. Wspominamy o stylu zalecanym przez społeczność zarządzającą rozwojem języka (Python Software Foundation), choć dość dużą popularność zdobyły też alternatywne zalecenia Google Python Style Guide (http://google.github.io/styleguide/pyguide.html). Różnice pomiędzy nimi są sprawą drugorzędną, a w obu przypadkach główny cel jakim jest uporządkowanie naszej pracy programistycznej zostanie osiągnięty. Oczywiście podobne konwencje mogą obowiązywać w ramach firmy czy nawet zespołu. W takim wypadku należy oczywiście stosować konwencje własnej organizacji (co można interpretować jako odzwierciedlenie ogólnej zasady PEP-20: zachowuj czytelność kodu — *Readability counts* ale także w myśl łacińskiej paremii: zasada szczegółowa uchyla zasadę ogólną — *Lex specialis derogat legi generali*). Najważniejsze zasady PEP-8, z punktu widzenia początkującego programisty, można podsumować następująco:

- nazwy zmiennych i funkcji powinny składać się wyłącznie z małych liter, możliwe jest stosowanie znaku '_' do oddzielenia słów; właściwe — zgodne ze stylem — nazwy to np. listaplikow albo lista_plikow, natomiast nazwy takiej jak ListaPlikow albo listaPlikow nie są zalecane,
- użycie 4 spacji jest preferowane niż tabulator (natomiast zastosowanie konwencji mieszanej: czasem spacje, czasem tabulator nie zostanie właściwie zinterpretowane przez Python 3 i jest niedozwolone),
- stosuj zapis plików źródłowych zgodny z UTF-8 (w Python 3 jest to domyślny format kodowania znaków i nie jest wymagana deklaracja strony kodowej UTF-8),
- staraj się, żeby długość linii kodu nie przekraczała 79 znaków (w wyjątkowych sytuacjach dopuszcza się maksymalną długość linii 99 znaków).

Osobna grupa zaleceń PEP-8 dotyczy stosowania białych znaków w środku linii (a więc takich, które nie mają znaczenia dla sposobu działania kodu).

- Umieszczaj spacje pomiędzy operatorami; dobry styl to: a = b + c, a = b * c / x % y, a mniej czytelne (choć równoważne dla interpretera): a=b+c, a=b*c/x%y,
- Jeśli używasz operatorów o różnym priorytecie wykonania (zob. Tab. 1.1), wówczas można zwiększyć czytelność kodu gdy nie umieścimy spacji wokół operatorów o wyższym priorytecie, np. a = b*c + z/x jest preferowane wobec np.: a = b * c + z / x albo a = b*c+z/x, choć w tym przypadku przewaga jednego nad drugim jest sprawą bardzo subiektywną,
- Nie stosuj spacji po nawiasie otwierającym ani przed nawiasem zamykającym („przypinaj" nawiasy), dobry styl to: a = (b + c), a mniej czytelny: a = (b + c),

– W przypadku wyrażeń dłuższych niż jeden wers[11], stosuj „łamanie linii" przed znakiem operatora, przykładowo bardziej czytelne:

```
dluga_formula = (pierwsza_dluga_zmienna
          + druga_dluga_zmienna)
          *  trzecia_dluga_zmienna
```

a mniej czytelne:

```
dluga_formula = (pierwsza_dluga_zmienna +
          druga_dluga_zmienna) *
          trzecia_dluga_zmienna
```

– Używaj dwu pustych linii przed i po: każdej definicji funkcji lub klasy oraz za blokiem instrukcji 'import'.

1.9 Dalsze lektury i możliwe „ścieżki" uczenia

Zastosowanie Python są bardzo szerokie, tak więc nasza książka otwiera przed Czytelnikami wiele możliwości przetwarzania danych i pokazuje przykłady badań i pomysłów jak „ułatwić sobie życie" stosując Python. Po przećwiczeniu podanych przykładów, pojawi się więcej pomysłów i mamy nadzieję chęć pogłębienia Python w wybranym obszarze.

Dlatego poświęcamy ten fragment na prezentację dalszych możliwości oraz pokazanie różnych ścieżek rozwoju (kariery) z użyciem Python, a w ślad za tym bardziej zaawansowanych lektur i materiałów.

Wyobraźmy sobie, że stworzyliśmy ciekawy program do przetwarzania danych. Może to być program do łączenia wielu plików Excela (zob. rozdz. 2, dodatek) albo program pobierający dane z serwisów ogłoszeniowych (zob. rozdz. 3.2). Mało kto chciałby ręcznie wpisywać ręcznie nazwy plików czy adres serwisu i wskazówki do wyszukania treści w nim zawartych do skryptu Pythona. Dlatego przewidując użycie tego programu przez osoby nie znające Python — będziemy zmuszeni wprowadzić jakąś formę przyjaznego (dla człowieka) interfejsu graficznego (GUI — Graphic User Interface). W zespołach programistycznych taką rolę pełni *front-end* developer. Przy czym interfejs może wystąpić albo formie klasycznego programu uruchamianego z naszego komputera albo aplikacji webowej, działającej na serwerze ale uruchamianej z przeglądarki internetowej końcowego użytkownika. W pierwszym przypadku użyjemy środowiska graficznego TkInter (od wersji 3.7 dołączanego do standardowych dystrybucji Python) lub PyQt — kompleksowej biblioteki z zestawem narzędzi do budowy interferjsu graficznego (popularność zyskała szczególnie wcześniejsze wersje dla języków C++ i Java). bardzo popularny wśród programistów C++ i Java. W przypadku coraz bardziej popularnych aplikacji webowych Py-

[11] „Łamanie" linii przy długich wyrażeniach pokazaliśmy w rozdz. 1.7.

thon oferuje jeszcze więcej możliwości. Dla początkujących programistów i niezbyt rozbudowanych serwisów, dobre są mikro-framemorki np. Flask czy Bottle. W momencie pisania tej książki (2020), sporą popularność zyskiwał Pyramid, choć w Polsce grono jego użytkowników jest niewielkie. W przypadku dużych serwisów, z obszerną zawartością i rozbudowaną obsługą użytkowników (panel logowania, administracji, poziomy uprawnień, zarządzanie treścią), konieczne będzie użycie bardziej kompleksowej biblioteki jaką jest Django.

Data scientist („badacz danych") został w 2012 roku uznany przez miesięcznik Harvard Business Review jako jeden z najbardziej „seksownych" zawodów na świecie. Oczywiście trudno powiedzieć ile w tym przesady, ale z pewnością trzeba przyznać, że uczenie maszynowe i sztuczna inteligencja są bardzo przyszłościowe. O uczeniu maszynowym piszemy w rozdz. 5, ale przedstawiamy tam wyłącznie podstawowe metody uczenia nadzorowanego[12]. Znajomość ekonometrii, statystyki i matematyki (zwłaszcza algebry liniowej) w połączeniu z umiejętnością programowania w Python (ewentualnie innym języku programowania wysokiego poziomu) otworzy drogę do kariery data scientist. Czytelnicy zainteresowani tym tematem powinni w pierwszej kolejności odświeżyć sobie wspomniane już przedmioty „matematyczne". Dla osób, które zaczynają naukę statystyki i ekonometrii prawie od zera, polecamy podręcznik Aczela (2017). Jeśli chodzi o Python, z pewnością trzeba zacząć od zaprzyjaźnienia się z modułami *numpy* i *scipy*. Biblioteki te, w połączeniu z *pandas* (omówionym w rozdz. 2.8) i *sklearn* (wybrane metody omówione w rozdz. 5) pozwolą rozpocząć przygodę z data science.

Pewne problemy uczenia maszynowego wymagają sięgnięcia do bardziej skomplikowanych metod uczenia głębokiego (lub inaczej: uczenia sztucznych sieci neuronowych). Przykładowe zastosowania tych problemów to: computer vision (rozpoznawanie i przetwarzanie obrazów i wideo) oraz sterowanie pojazdami autonomicznymi. Dla osób z gruntownym przygotowaniem matematycznym, zainteresowanych teorii uczenia głębokiego (wykorzystującego wielowarstwowe sieci neuronowe) można polecić przetłumaczoną na polski książkę Bengio, Courville i Goodfellow (2018). Od strony języka Python, warto zapoznać się z modułem tensorflow-keras (np. https://www.tensorflow.org/guide), a także z szerszym środowiskiem obliczeniowym *tensorflow*.

[12] Wprowadzenie do pozostałych metod uczenia maszynowego (uczenie nienazdorowane, analiza sieciowa i metody asocjacji) można znaleźć w: Provost i Fawcett (2019) oraz Boschetti i Massaron (2017).

2. Podstawowe typy i struktury danych

Rozdział poświęcamy omówieniu podstawowych typów danych. Przedstawione zostaną typy służące do reprezentacji danych liczbowych: liczb całkowitych — *int* oraz liczb zmiennoprzecinkowych — *float*, typ napis odpowiadający ciągowi znaków oraz typ *bool* reprezentujący wartości logiczne. Opiszemy także takie struktury jak lista, krotka, słownik oraz zbiór, służące do przechowywania elementów dowolnego typu. Opiszemy ponadto podstawowe operacje na wyżej wymienionych strukturach.

2.1 Typy liczbowe

W celu reprezentacji liczb Python wykorzystuje głównie dwa typy zmiennych: *int* dla liczb całkowitych i *float* dla liczb zmiennoprzecinkowych. Być może zaskoczy to Czytelnika, że przykładowo liczby 3 oraz 3.0 (pamiętamy, że funkcję separatora dziesiętnego pełni kropka, a nie przecinek) są inaczej przechowywane. Do sprawdzenia typu zmiennej służy funkcja *type()*, która wywołana z jednym parametrem (obiektem) zwróci jego typ. Rezultaty wywołań funkcji *type()* dla liczb 3 oraz 3.0 przedstawiono poniżej.

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
```

Int (*integer*) służy do przechowywania liczb całkowitych. Co warto podkreślić, jest ich dokładną reprezentacją (w przeciwieństwie do już wcześniej wspomnianych liczb zmiennoprzecinkowych), a wielkość liczb całkowitych jest ograniczona jedynie możliwościami pamięci komputera.

Klasa *float* oznacza liczby zmiennoprzecinkowe (zmiennopozycyjne) — w praktyce są to te, które zawierają kropkę (wspomniany już przykład liczby 3.0), choć mogą być także zapisane z wykorzystaniem tzw. notacji naukowej, np.:

```
>>> 0.3e2   # alternatywnie można użyć do zapisu 0.3E2
30.0
>>> 0.3e-2
0.003
```

Co ważne, istnieje ograniczony zakres liczbowy ich reprezentacji. Wartości większe od górnego zakresu, równego w przybliżeniu 1.8×10^{308}, są traktowane jako nieskończone (*inf — infinity*), z kolei liczby odpowiednio małe, traktowane są jako 0. Pokazuje to kolejny przykład:

```
>>> import sys
>>> sys.float_info.max #  największa wartość typu float
1.7976931348623157e+308
>>> 1.7976931348623159e+308 # zmiana w ostatniej cyfrze
mantysy
inf
>>> 1 / 1.7976931348623159e+308
0.0
```

Należy również pamiętać, że nie zawsze możliwa jest dokładna reprezentacja liczby dziesiętnej w systemie binarnym, przykładowo:

```
>>> a = 0.7
>>> print(' %.15f.' %a)  # %.15f. oznacza 15 miejsc po prze-
cinku
0.700000000000000.
>>> print(' %.25f.' %a)  # %.25f. oznacza 25 miejsc po prze-
cinku
0.6999999999999999555910790.
>>> print(' %.35f.' %a)  # %.35f. oznacza 35 miejsc po prze-
cinku
0.69999999999999995559107901499373838.
```

Widzimy zatem, że przechowywana wartość jest równa 0.7 z pewnym przybliżeniem (w praktyce jest to najbliższa liczba, którą ma dokładną reprezentację binarną).

Inną specjalną wartością jest *nan* (*not a number*), którą możemy otrzymać na przykład jako wynik działania:

```
>>> a = 1.7976931348623159e+308
>>> a
inf
```

```
>>> a / a
nan
```

Uważa się, że typ logiczny *bool* również należy do wartości liczbowych. Przyjmuje on tylko dwie wartości — prawda — *True* oraz fałsz — *False*. Jest to typ zwracany przy dokonywaniu testów logicznych, w szczególności przez operator porównania:

```
>>> 3 > 2
True
>>> 3 < 2
False
```

Wartości *True* i *False* służą m. in. rozpoznawaniu czy obiekty „złożone" są puste (np. listy, napisy) lub są zerem (w przypadku liczb) — wówczas obiekt jest fałszem. Inaczej mówiąc obiekt/liczba jest prawdą jeśli jest niepusty/przyjmuje niezerową wartość. W poniższych przykładach sprawdzamy jaka jest wartość logiczna liczby 0 oraz obiektu lista, który jest omówiony w rozdziale 2.3. Zachęcamy Czytelnika do własnych eksperymentów z innymi obiektami, omówionymi w rozdziałach 2.2–2.6.

```
>>> bool(0)      # wartość logiczna liczby zero
False
>>> bool([0])   # lista zawierająca 0 jest niepusta
True
>>> bool([])     # wartość logiczna pustej listy
False
```

Wartość fałsz przypisana jest również szczególnej wartości — *None*, która jest odpowiednikiem obiektu pustego. Wspominamy o niej, choć nie będzie miała dla nas większego praktycznego znaczenia. Wartość tę zwraca na przykład funkcja **print**:

```
>>> a = 6
>>> b = print(a)
6
>>> print(b)
None
```

Typowanie w Pythonie jest dynamiczne, co znaczy, że dostosowuje się automatycznie do typów przechowywanych wartości i nie wymaga zadeklarowania typu wprost, o czym była mowa w rozdziale 1.

Co więcej, można wykonywać niektóre operacje na typach mieszanych, np. otrzymamy poprawny wynik dodawania liczby całkowitej i zmiennoprzecinkowej:

```
>>> a = 2
>>> b = 3.3
```

```
>>> a + b
5.3
```

Python dokonuje bowiem konwersji dodawanych liczb na typ zmiennoprzecinkowy i przeprowadza na nim obliczenia. Takiej samej automatycznej konwersji dokonuje Python w przypadku porównań, dlatego rezultatem polecenia *3 == 3.0* jest wartość *True*, czyli prawda.

W celu dokonania (jawnej) konwersji możemy użyć tzw. rzutowania wskazując pożądany typ i w nawiasie podając zmienną/wartość, np.:

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> float(a)
2.0
>>> b = float(a)
>>> type(b)
<class 'float'>
>>> b
2.0
```

W powyższym przykładzie w zmiennej a przechowujemy liczbę 2 (całkowitą). Zmienna jest zatem typu *int*. Następnie tworzymy zmienną b dokonując rzutowania zmiennej a na typ *float*. W wyniku tej operacji w zmiennej b przechowywana jest wartość 2.0, która jest typu *float*. Czytelnik może spotkać się z operacją konwersji obiektów na typ napis (przedstawiony w rozdz. 2.2) na przykład podczas operacji zapisu danych do pliku tekstowego.

2.2 Napisy

Czytelnik poznał już podstawowe typy danych służących do przechowywania liczb. Pojawia się zatem pytanie jak przechowywać napisy (fragmenty tekstu, łańcuchy znaków)? Służy do tego osobna klasa *str* (od angielskiego słowa *string*). Aby uzyskać zmienną tego typu, należy ująć napis w znaki `" "` lub `' '`, np.:

```
>>> type('zegar')
<class 'str'>
>>> type("zegar")
<class 'str'>
>>> 'zegar' == "zegar"
True
```

Możliwe jest ponadto wprowadzanie komentarzy wieloliniowych ograniczając je znakami `' ' '` (lub `" " "`).

```
>>> ''' to jest bardzo długi
... wieloliniowy
... napis
... '''
' to jest bardzo długi\nwieloliniowy\nnapis\n'
```

Okazuje się, że Python zwrócił powyższy napis w jednej linijce, oddzielając kolejne linie znakami `\n`. Jest to tzw. znak specjalny, który oznacza początek nowej linii. Zauważmy, że „specjalne działanie" litery n zostało uaktywnione znakiem \, czyli odwrotnego ukośnika (*backslash*). Znak ten zmienia interpretację znaku (znaków) następujących po nim i dlatego bywa nazywany znakiem modyfikacji bądź ucieczki (*escape character*). Sposoby deklaracji napisu z wykorzystaniem znaków `''` (`""`) pozwalają w łatwy sposób tworzyć napisy zawierające znaki `""` (`''`). Do tematu znaków specjalnych, a także możliwości zapisu typu str wrócimy w rozdziale 4, gdzie Czytelnik znajdzie szersze ich omówienie.

Napis jest typem iterowalnym, co oznacza, że możemy odwołać się do poszczególnych jego elementów (znaków) stosując indeksowanie liczbami całkowitymi, przy czym, jak już wiadomo z rozdziału 1.1, pierwszy znak napisu ma numer zerowy. Zwracana wartość (a także pusty napis) należy również do klasy *str*:

```
>>> "zegar"[0]
'z'
>>> type("zegar"[0])
<class 'str'>
```

Wykorzystując nawias kwadratowy możemy również odwołać się do dłuższych (niż jednoelementowy) fragmentów (wycinków) (*slice*) napisów podając indeks początkowy oraz końcowy fragmentu rozdzielone dwukropkiem, czyli stosując składnię `napis[początek:koniec]`, np.:

```
>>> "abcd"[1:3]
'bc'
```

Dlaczego Python zwrócił taki wynik? Indeksowanie rozpoczyna się od 0, dlatego litera oznaczona numerem 1 to b, kolejnej literze — c przypisany jest indeks 2, z kolei indeks 3 odnosi się do litery d. Wywołanie [1:3] zwróciło zatem część napisu poczynając od znaku z indeksem początkowym (włącznie) do znaku z indeksem końcowym (ale już bez tej litery). Można to porównać do matematycznego zapisu przedziału lewostronnie domkniętego i prawostronnie otwartego.

a	b	c	d
0	1	2	3

Aby lepiej zrozumieć tę regułę prześledźmy kolejne przykłady:

```
>>> "abcd"[1:2]
'b'
```

Otrzymaliśmy jedynie znak odpowiadający indeksowi początku wycinka, ponieważ kolejna litera ma już indeks 2, czyli odpowiadający końcowi podanego przedziału, a zatem nie zostanie zwrócona.

```
>>> "abcd"[0:3]
'abc'
```

Fragment trzyliterowy zwrócony w tym przypadku odpowiada kolejnym indeksom: 0 (litera a), 1 (b), 2 (c). Litera odpowiadająca indeksowi końca, czyli d, została pominięta. Jest to ogólna zasada, którą Czytelnik miał okazję poznać w rozdziale 1.6 przy opisie sekwencji range.

Możliwe jest niepodawanie pełnego zakresu — wskazanie tylko początku fragmentu — takie wywołanie zwróci podnapis począwszy od znaku o wskazanym indeksie początku (w naszym przykładzie 1) do końca napisu, czyli tak, jakby pomięty indeks końca odpowiadał długości napisu, rozumianej jako liczba jego znaków. Długość napisu „abcd" wynosi 4, dlatego poniższe wywołania są tożsame:

```
>>> "abcd"[1:]
'bcd'
>>> "abcd"[1:4]
'bcd'
```

Możliwe jest ponadto wskazanie jedynie indeksu końca, które zadziała w ten sposób, że potraktuje np. `"abcd"[:4]` tak samo jak `"abcd"[0:4]`, czyli tym razem za pominięty indeks początku przyjmie wartość zero — tzn. początek napisu. Możliwe jest również odwołanie do fragmentu napisu bez podawania indeksów początku i końca, np. `"abcd"[:]`. Przyjmie ono wówczas domyślne wartości odpowiednio początku i końca i zwróci cały napis.

Omówiona powyżej składnia umożliwia tworzenie fragmentów znak po znaku, ale możliwe jest również tworzenie fragmentów „z przesunięciem"/„z krokiem"). Wówczas wywołanie jest postaci: `napis[początek:koniec:krok]`, np. `"abcd"[0:3:2]` zwróci podnapis 'ac', tzn. z fragmentu odpowiadającemu indeksowi początku i końca — w powyższym przykładzie od 0 do 3, czyli 'abc' zwróci co drugi znak, czyli zwróci a, pominie b, zwróci c.

Możliwe jest także indeksowanie od końca, przy czym, aby uzyskać ostatni element napisu, należy użyć indeksu -1:

```
>>> "abcd"[-1]
'd'
```

Warto zapamiętać regułę, że wykorzystanie ujemnego indeksu jest równoważne wykorzystaniu indeksu dodatniego równego długości napisu skorygowanej o ujemny indeks. W przykładzie powyżej długość napisu to 4, a zatem wywołanie "abcd"[-1] odpowiada dokładnie wywołaniu z indeksem 4-1=3, czyli "abcd"[3].

Należy ponadto zwrócić uwagę na kolejność podawania indeksów — zakres od ostatniego do przedostatniego elementu jest pusty, zatem takie wywołanie zwróci pusty podciąg, natomiast od zakres od przedostatniego do ostatniego elementu zwróci literę c:

```
>>> "abcd"[-1:-2]
''
>>> "abcd"[-1:-2] == ""
True
>>> "abcd"[-2:-1]
'c'
```

Możliwe jest wreszcie tworzenie podnapisów z ujemnym przesunięciem, przykładowo, aby wypisać napis od końca, należy odwołać się do całego zakresu wskazując krok -1:

```
>>> "abcd"[::-1]
'dcba'
```

Warto zaznaczyć, że napis jest typem niezmienniczym (*immutable*), co oznacza, że nie możemy zmieniać jego wartości, o czym zostaniemy poinformowani w komunikacie o błędzie w przypadku próby jego modyfikacji:

```
>>> "abcd"[0] = 'k'
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    "abcd"[0] = 'a'
TypeError: 'str' object does not support item assignment
```

Możemy jednak stworzyć inny obiekt, który będzie miał pożądaną przez nas wartość:

```
>>> "kbcd"
'kbcd'
```

W przypadku użycia zmiennych (nazw), istnieje możliwość zmiany poprzez zmianę obiektu, z którą dana nazwa (zmienna) jest powiązana. Przykładowo przypisanie `napis = "abcd"` tworzy obiekt typu *str* i przechowuje w nim wartość 'abcd'. Jednocześnie łączy ten obiekt ze zmienną napis. Zmiana wartości zmiennej napis w sposób `napis[0] = "k"` zwróci komunikat o błędzie, ale możliwe jest połączenie zmiennej napis z innym obiektem w pamięci, zawierającym wartość `"kbcd"`:

```
>>> napis = "kbcd"
>>> napis
'kbcd'
```

Dla napisów możliwe są takie działania jak ich dodawanie (nazywane też konkatenacją) oraz mnożenie przez liczbę naturalną (zwielokrotnienie). Konkatenacja to połączenie dwóch lub więcej tych samych lub różnych napisów w jeden długi napis. Zwielokrotnienie to kilkukrotna konkatenacja tego samego napisu. Wyjaśniają to następujące przykłady:

```
>>> "*" + "abcd" + "abcd" + "*"
'*abcdabcd*'
>>> "*" + "abcd"*2 + "*"
'*abcdabcd*'
```

Napis jest elementem iterowalnym — w przykładzie poniżej będziemy wypisywać każdą kolejną literę:

```
>>> napis = "abcd"
>>> for i in napis:
...     print(i)
a
b
c
d
```

Alternatywnie, aby uzyskać ten sam rezultat, moglibyśmy wykorzystać wiedzę dotyczącą indeksowania i użyć komendy:

```
>>> for i in range(len(napis)):
...     print(napis[i])
...
a
b
c
d
```

Zwracamy jednak uwagę, że możliwość iteracji napisu bez konieczności odwoływania się wprost do indeksów upraszcza składnię, przez co jest w duchu Pythona (*Pythonic*) i jest szybsza.

Więcej informacji dotyczących przetwarzania napisów przedstawiono w rozdziale 4.

2.3 Lista

Lista jest ciągiem elementów dowolnego typu. Podobnie jak napis jest typem indeksowanym, lecz w odróżnieniu od niego, pozwala na modyfikację. Do deklaracji listy służą nawiasy kwadratowe, poniżej zadeklarujemy pustą listę:

```
>>> lista = []
>>> type(lista)
<class 'list'>
```

Możemy również zadeklarować listę wymieniając wewnątrz nawiasów kwadratowych jej elementy oddzielone przecinkami[1]:

```
>>> lista = [1, 2, 3, 4, 5]
>>> lista
[1, 2, 3, 4, 5]
```

Odwołanie do poszczególnych elementów listy, podobnie jak w przypadku napisu, umożliwiają nawiasy kwadratowe. Zmieńmy teraz pierwszy element listy (liczbę 1) na liczbę 11. Pamiętamy z rozdziału 1.1, że Python indeksuje od zera, dlatego pierwszy element listy ma indeks 0.

```
>>> lista[0] = 11
>>> lista
[11, 2, 3, 4, 5]
```

Widzimy, że zmiana powiodła się. Możliwe jest również, analogiczne do napisów, tworzenie fragmentów listy, jak również indeksowanie „od końca":

```
>>> lista[0:3]
[1, 2, 3]
>>> lista[0:3:2]
[1, 3]
>>> lista[-1]
5
```

Jak już wspominano, w odróżnieniu od napisów, lista jest typem pozwalającym na modyfikację. Innymi słowy możemy zmieniać poszczególne elementy listy, w tym je usuwać, a także dołączać nowe. Co ciekawe, elementy tej samej listy mogą być różnych typów, w szczególności elementami listy mogą być inne listy (lub typy złożone np. słowniki zob. rozdz. 2.5). Dodajmy zatem do naszej listy kolejne elementy: listę, napis oraz liczbę zmiennoprzecinkową. Do dodawania elementów do listy służą trzy metody: **.append**(), **.extend**() oraz **.insert**(). Prześledźmy ich działanie na następujących przykładach:

[1] Dopuszcza się także umieszczenie przecinka po ostatnim elemencie listy np. [1, 2, 3, 4, 5,].

```
>>> lista
[11, 2, 3, 4, 5]
>>> lista.append([0, 1, 2])
>>> lista
[11, 2, 3, 4, 5, [0, 1, 2]]
>>> lista.extend([0, 1, 2])
>>> lista
[11, 2, 3, 4, 5, [0, 1, 2], 0, 1, 2]
>>> lista.append("kot")
>>> lista
[11, 2, 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot']
>>> lista.extend("kot")
>>> lista
[11, 2, 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot', 'k', 'o', 't']
```

Metody .**append**() i .**extend**() dodają nowe elementy na końcu listy, przy czym .**extend**() operuje na argumentach iterowalnych (w powyższych przykładach liście [0, 1, 2] i napisie „kot") i dołącza je do istniejącej listy po jednym elemencie (odpowiednio liczbie w pierwszym przypadku i literze — w drugim). Metoda .**append**() operuje na dowolnym argumencie (iterowalnym lub nie) i dołącza go (w całości) na końcu istniejącej listy.

Metoda .**insert**() służy do wstawienia elementu na określoną pozycję listy (a więc w odróżnieniu od .**append**() gdzie dodawaliśmy element na koniec listy). Metoda ta wymaga podania dwóch argumentów — indeksu i elementu. Indeks służy do przekazania informacji o miejscu dołączenia nowego elementu — element zostanie dołączony przed wskazanym indeksem, czyli w przypadku wskazania indeksu zerowego — na początku listy.

```
>>> lista.insert(0, "pies")
>>> lista
['pies', 11, 2, 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot', 'k',
'o', 't']
```

Ciekawostka: Aby odwołać się do listy będącej siódmym elementem naszej listy (tj. o indeksie 6), użyjemy składni `lista[6]`, natomiast, aby odwołać się np. do zera stanowiącego pierwszy element tej listy, użyjemy `lista[6][0]`.

Do usuwania elementów z listy służą metody .**pop**() i .**remove**(). Metoda .**pop**(*indeks*) pozwala na wskazanie indeksu i zwraca element przypisany temu indeksowi, jednocześnie usuwając go z listy. W poniższym przykładzie usuniemy element o indeksie 1 (tj. drugi licząc od początku listy).

```
>>> lista.pop(1)
11
>>> lista
['pies', 2, 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot', 'k', 'o',
't']
```

Metoda .**pop**() wywołana bez żadnego argumentu usuwa i zwraca ostatni element. Sprawdzenie rezultatów jej wywołania pozostawiamy Czytelnikowi.

Metoda .**remove**() wymaga podania elementu do wykasowania, po czym usuwa go z listy. W przypadku, gdy występuje więcej niż jeden taki element, metoda ta usuwa pierwsze jego wystąpienie. W przykładzie poniżej usunięta zostanie pierwsza liczba 2.

```
>>> lista
['pies', 2, 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot', 'k', 'o',
't']
>>> lista.remove(2)
>>> lista
['pies', 3, 4, 5, [0, 1, 2], 0, 1, 2, 'kot', 'k', 'o', 't']
```

Podczas zawodów jazdy figurowej wylicza się średnią z not 7 z 9 sędziów (odrzuca się noty skrajne). Taki system oceny możemy zaimplementować przy użyciu listy ocen, z której usuwa się elementy o wartości maksymalnej i minimalnej. Dopiero z takiej „okrojonej" listy ocen oblicza się średnią arytmetyczną, która służy do opracowania listy rankingowej.

```
>>> oceny = [9, 10, 9.5, 9, 8, 6.5, 9, 8, 8.5]
>>> najwieksza_ocena = max(oceny)
>>> oceny.remove(najwieksza_ocena)
>>> najmniejsza_ocena = min(oceny)
>>> oceny.remove(najmniejsza_ocena)
>>> print(sum(oceny) / len(oceny))
8.714285714285714
```

Listy możemy również łączyć za pomocą operatorów + (konkatenacja lub prościej mówiąc złączanie list) oraz * (zwielokrotnianie). Zasada ich działania jest analogiczna do działań zdefiniowanych dla napisów, np.:

```
>>> lista = [2, 3, 4]
>>> lista + lista
[2, 3, 4, 2, 3, 4]
```

W rezultacie otrzymujemy „długą" listę stanowiącą sumę list, czyli inaczej jedną, długą listę, obejmującą elementy składowe obu list sumowanych, zachowując kolejność elementów. Działanie operatora + dla list polega na łączeniu (konkatenacji) list, a nie na dodawaniu (liczb) (tego typu operacje możliwe są na obiektach Series i DataFrame z modułu *pandas*, co omówimy w rozdz. 2.8).

Możliwe jest również mnożenie list przez liczbę naturalną, np.:

```
>>> lista
[2, 3, 4]
>>> lista * 2
[2, 3, 4, 2, 3, 4]
```

Również w tym przypadku otrzymaliśmy długą listę, składającą się z dwóch list „początkowych", zachowując kolejność ich elementów.

Do przeszukiwania list użyteczne są metody: .**index**(), .**count**() oraz słowo kluczowe *in*. Załóżmy, że chcemy sprawdzić czy liczba 3 jest elementem naszej listy. Wywołując metodę .**index**(element) wraz z szukanym elementem otrzymamy indeks wskazujący, że liczba 3 pojawia się po raz pierwszy na naszej liście na drugim miejscu. Wywołanie `lista.`**count**(3) zwróci natomiast informację o dwukrotnym wystąpieniu liczby 3 na naszej liście.

```
>>> lista = [2, 3, 4, 2, 3, 4]
>>> lista.index(3)
1
>>> lista.count(3)
2
```

W praktyce, aby sprawdzić czy dany element należy do listy będziemy używać słowa kluczowego *in* — aby sprawdzić czy warunek *`element in lista`* jest prawdziwy. Inaczej mówiąc zwrócona wartość logiczna wskaże nam czy element jest obecny na liście czy nie.

```
>>> 3 in lista
True
>>> 10 in lista
False
```

Mamy również dostępną możliwość sortowania elementów:

```
>>> lista.sort()
>>> lista
[2, 3, 4, 5, 11]
```

Do sortowania Python używa algorytmu *Timsort* zaimplementowanego przez Tima Petersa (Peters, 2002), a bazującego na idei przedstawionej przez Petera McIlroya (McIlroy, 1993, za: Zhang, Meng, Liang, 2016).

Możliwe jest sortowanie list przechowujących inne elementy, przykładowo sortowanie list przechowujących napisy zwróci listę o elementach uporządkowanych według kolejności alfabetycznej:

```
>>> lista = ["beta", "alfa", "aaa"]
>>> lista.sort()
>>> lista
['aaa', 'alfa', 'beta']
```

Jak już wspomniano, w Pythonie znaczenie ma wielkość liter i jeśli chodzi o sortowanie, to duże litery będą przed małymi, a liczby przechowywane jako napisy (tzn. wpisane w cudzysłowie) — przed literami, np.:

```
>>> lista = ["1" , "0" , "A" , "alfa" , "beta" , "Alfa" ,
"aLfa"]
>>> lista.sort()
>>> lista
['0', '1', 'A', 'Alfa', 'aLfa', 'alfa', 'beta']
```

Możliwe jest także sortowanie również listy list, co widać na kolejnym przykładzie:

```
>>> lista = [[1.2], [1], [-2], [2, 3], [0]]
>>> lista.sort()
>>> lista
[[-2], [0], [1], [1.2], [2, 3]]
```

Warto natomiast zaznaczyć, że możliwe jest sortowanie jedynie list zawierających elementy tego samego typu, a próba sortowania listy złożonej z różnych typów, przykładowo liczb całkowitych oraz napisów, zwróci komunikat o błędzie. W poniższej liście dwa pierwsze elementy to liczby wpisane bez cudzysłowu, a więc przechowywane jako liczby całkowite. Nie jest więc możliwym posortowanie listy. Wyświetlony komunikat informuje nas o braku możliwości dokonania porównania ‚<' pomiędzy typami *str* oraz *int*.

```
>>> lista = [1, 0, "A", "alfa", "beta", "Alfa", "aLfa"]
>>> lista.sort()
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    lista.sort()
TypeError: '<' not supported between instances of 'str' and
'int'
```

Listy składane (*list comprehension*) służą do przeprowadzania działań na poszczególnych elementach zbioru iterowalnego i mają ogólną składnię: `[działanie for element in typ_iterowalny`[2]`]`. W przykładach poniżej będziemy używać składni `[działanie for element in lista]`. Użycie nawiasów kwadratowych informuje nas o tym, że typem zwracanym będzie lista, natomiast wyrażenie wewnątrz nawiasu — że elementami składowymi będą rezultaty pewnej operacji („działanie"), przeprowadzonej dla każdego elementu z danego typu iterowalnego.

Przykładowy kod generuje drugie potęgi kolejnych elementów listy.

```
>>> lista
[2, 3, 4, 2, 3, 4]
>>> [element ** 2 for element in lista]
[4, 9, 16, 4, 9, 16]
```

[2] Typy iterowalne są złożone z elementów ułożonych w określonej kolejności (np. lista, napis, krotka), choć mogą być to również typy, w których kolejność elementów nie jest ustalona (np. słownik, zbiór).

Możliwe jest uzyskanie tego samego rezultatu w inny sposób — iterując po elementach listy, przy czym kwadraty poszczególnych elementów są dodawane do listy o nazwie kwadraty metodą .**append**():

```
>>> lista = [2, 3, 4, 2, 3, 4]
>>> kwadraty = []
>>> for i in lista:
...     kwadraty.append(i ** 2)
>>> kwadraty
[4, 9, 16, 4, 9, 16]
```

Alternatywnie moglibyśmy iterować nie po elementach, ale po indeksach listy jak w kodzie poniżej:

```
>>> lista = [2, 3, 4, 2, 3, 4]
>>> kwadraty = []
>>> for i in range(len(lista)):
...     kwadraty.append(lista[i] ** 2)
>>> kwadraty
[4, 9, 16, 4, 9, 16]
```

Wszystkie wymienione metody zwracają ten sam rezultat, ale tylko pierwsza z nich (*list comprehension*) jest w duchu Pythona (a więc *Pythonic*) i jest polecana ze względu na prostotę i czytelność zapisu oraz wydajność (np. Jaworski, Ziadé, 2017, s. 58–59).

Nie są to wszystkie możliwości list złożonych — możliwe jest rozszerzenie składni o sprawdzenie warunku i wykonanie działań na kolejnych elementach składowych listy, spełniających podany warunek. Wówczas składnia takiego wyrażenia ma postać `[działanie for element in lista if warunek]`. Elementami składowymi zwracanej listy będą rezultaty pewnej operacji (`działanie`), przeprowadzonej dla każdego elementu z danej listy (`lista`), ale tylko wówczas jeśli spełni on określony `warunek`.

Przykładowy kod sprawdza czy element z listy jest podzielny przez 2, a jeśli tak — dzieli go przez 2. Na naszej liście mamy cztery elementy spełniające zadany warunek, stąd też otrzymamy na wyjściu dwie liczby 1, powstałe z dzielenia 2 przez 2 oraz dwie liczby 2 powstałe z dzielenia 4 przez 2.

```
>>> lista
[2, 3, 4, 2, 3, 4]
>>> [element / 2 for element in lista if element % 2 == 0]
[1.0, 2.0, 1.0, 2.0]
```

Zwracamy uwagę, że działanie może polegać na „przepisaniu" elementu listy bez jego modyfikacji, przykładowo, aby wypisać liczby podzielne przez dwa z naszej listy użylibyśmy składni:

```
>>> [element for element in lista if element % 2 == 0]
[2, 4, 2, 4]
```

Ciekawostka: Python jest nazywany językiem z „załączonymi bateriami" (*batteries included*), co oznacza, że dostarcza on dużą liczbę narzędzi gotowych do użycia (które w innych językach mogą wymagać samodzielnego zakodowania). Takimi funkcjami są na przykład funkcje **min**(), **max**() oraz **sum**():

```
>>> min([100, -2, 300, 2])
-2
>>> max([100, -2, 300, 2])
300
>>> sum([100, -2, 300, 2])
400
```

Jak już wspomniono, wartości logiczne służą m.in. do sprawdzenia czy obiekt przyjmuje wartość zerową (lub odpowiadającą zeru). W poniższym przykładzie przypisujemy zmiennej a wartość 0, a następnie sprawdzamy jej wartość logiczną:

```
>>> a = 0
>>> bool(a)
False
```

Analogiczną wartość uzyskamy w przypadku pustego napisu:

```
>>> a = ''
>>> bool(a)
False
```

Analogicznie jest w przypadku pustych zbiorów, list, słowników czy zakresów. Dlatego też funkcja **any**(), operująca na dowolnym obiekcie iterowalnym (np. na liście) zwróci wartość *True*, jeśli co najmniej jeden jej element przyjmuje tę wartość, a *False* w przypadku kiedy wszystkie jej elementy przyjmują wartość *False* lub w przypadku pustego obiektu:

```
>>> any(["", 0, None])
False
>>> any(["", 0, None, True])
True
```

Funkcja **all**() zwraca wartość prawda jeśli wszystkie elementy obiektu iterowalnego przyjmują wartość *True* oraz w przypadku kiedy obiekt interowalny jest pusty.

```
>>> all([0, 1, 2])
False
>>> all([1, 2])
True
```

```
>>> all([])
True
```

2.4 Krotka

Krotka (*tuple*) jest niemodyfikowalną listą, a do jej deklaracji używane są nawiasy okrągłe (). Możliwa jest również notacja z wykorzystaniem samych przecinków. Załóżmy, że chcemy otrzymać krotkę składającą się z trzech liczb: 1, 2 oraz 3. Poniższe sposoby jej utworzenia są tożsame:

```
>>> krotka = (1, 2, 3)
>>> krotka
(1, 2, 3)
>>> krotka = 1, 2, 3
>>> krotka
(1, 2, 3)
```

Krotka, podobnie jak lista jest typem iterowalnym — w poniższym przykładzie iterujemy po jej elementach i wypisujemy je:

```
>>> krotka = (1, 2, 3, 4)
>>> for i in krotka:
...     print(i)
...
1
2
3
4
```

Możliwe są ponadto odwołania do poszczególnych elementów, podobnie jak w napisach oraz listach, do fragmentów, w tym również z wykorzystaniem indeksów ujemnych:

```
>>> print(krotka[1], krotka[-1])
2 4
>>> krotka[0:2]
(1, 2)
```

Ze względu na brak możliwości modyfikacji, typ ten nie będzie przez nas szerzej wykorzystywany, dlatego poprzestaniemy na powyższym, skrótowym jego opisie. Warto jednak wiedzieć, że możliwe jest jego rzutowanie (konwersja) na listę:

```
>>> krotka = (1, 2, 3, 4)
>>> lista = list(krotka)
>>> lista
[1, 2, 3, 4]
```

Ten sam rezultat możemy uzyskać z wykorzystaniem list składanych:

```
>>> [element for element in krotka]
[1, 2, 3, 4]
```

Ciekawostka: Niektóre typy w Pythonie (w szczególności listy i krotki) można rozpakowywać, czyli przekazywać ich poszczególne elementy i przypisywać je do nowych zmiennych lub np. przekazywać bezpośrednio do funkcji.

W poniższym przykładzie listę 3 kolorów rozpakujemy do 3 zmiennych. Zwracamy uwagę na składnię — nazwy zmiennych są rozdzielone przecinkami:

```
>>> kolory = ["czerwony", "zielony", "niebieski"]
>>> kolor1, kolor2, kolor3 = kolory
>>> kolor1
'czerwony'
>>> kolor2
'zielony'
>>> kolor3
'niebieski'
```

Poniżej rozpakowanie listy bezpośrednio do funkcji print.

```
>>> print(*kolory)
biały zielony niebieski
```

Dla porównania funkcja print wywołana dla tej samej listy, ale bez jej rozpakowania:

```
>>> print(kolory)
['biały', 'zielony', 'niebieski']
```

2.5 Słowniki

Słowniki to inaczej odwzorowania (*mapping*) lub tablice haszujące/asocjacyjne. Przechowują dane w parach `{klucz: wartość}`, przy czym kolejne pary podajemy oddzielone przecinkiem. Kluczem w słowniku może być element niezmienniczy (np. napis, choć nie tylko), natomiast wartością — dowolny obiekt. W słowniku, inaczej niż w liście, wartości powiązane są z kluczami, ale nie ma znaczenia ich kolejność. Skoro zatem wartości (obiekty) przechowywane są według kluczy, klucze muszą być unikalne.

Przykładowy słownik przechowuje informacje o liczbie poszczególnych słów w pewnym tekście:

```
>>> slownik = {"Ala": 1, "ma": 1, "psa": 0, "kota": 1}
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 1}
```

Zwracamy uwagę, że o ile do deklaracji krotki służyły nawiasy zwykłe, listy — kwadratowe, tak do deklaracji słowników wykorzystywane są nawiasy klamrowe, nadal jednak kolejne elementy zapisujemy po przecinku. Taki sam rezultat moglibyśmy uzyskać deklarując pusty słownik, dodając następnie kolejne elementy z wykorzystaniem składni `słownik[klucz] = wartość`, jak przykładach poniżej:

```
>>> slownik = {}
>>> slownik['Ala'] = 1
>>> slownik['ma'] = 1
>>> slownik['psa'] = 0
>>> slownik['kota'] = 1
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 1}
```

Alternatywnie moglibyśmy użyć składni `dict(klucz = wartość)`. Zwracamy uwagę, że w tej składni kluczy nie wpisujemy w cudzysłowach, pomimo że są napisami. Co więcej, ten sposób tworzenia słownika wymaga, aby kluczami były napisy.

```
>>> slownik = dict(Ala = 1, ma = 1, psa = 0, kota = 1)
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 1}
```

Odwołanie do konkretnego elementu słownika umożliwiają klucze oraz składnia `słownik[klucz]` jak w poniższych przykładach:

```
>>> slownik['kota']
1
```

Warto podkreślić, że odwoływanie się do elementu (wartości) przebiega za pomocą klucza, a nie np. jak w przypadku listy — pozycji elementu (indeksu).

Słownik możemy zmieniać, np.:

```
>>> slownik['kota'] += 1
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 2}
```

Aby dodać element do słownika wystarczy odwołać się do klucza nieistniejącego w modyfikowanym słowniku, a następnie przypisać mu wartość:

```
>>> slownik['chomika'] = 1
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 2, 'chomika': 1}
```

Kluczami słownika mogą być różne typy pod warunkiem, że są niezmiennicze:

```
>>> slownik[0] = "00"
>>> slownik
```

```
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 2, 'chomika': 1, 0:
'00'}
```

Do usuwania elementów służy funkcja **del** słownik[klucz] oraz metoda .**pop**(klucz), która analogicznie jak w przypadku list, usuwa element (o danym kluczu) i go zwraca.

```
>>> del slownik['chomika']
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 2, 0: '00'}

>>> slownik.pop(0)
'00'
>>> slownik
{'Ala': 1, 'ma': 1, 'psa': 0, 'kota': 2}
```

Jak już wspomnieliśmy, do odwoływania się do poszczególnych wartości zapisanych w słowniku służą klucze. Jeśli byśmy jednak chcieli odwołać się za pomocą klucza, który nie istnieje w słowniku, otrzymamy błąd:

```
>>> slownik['Ala']
1
>>> slownik['ala']
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    slownik['ala']
KeyError: 'ala'
```

Dlatego użyteczna jest metoda .**get**(klucz), która wywołana z kluczem, zwraca wartość do niego przypisaną, a jeśli klucz nie istnieje w słowniku zwraca wartość domyślną, którą jest *None*:

```
>>> slownik.get("Ala")
1
>>> slownik.get('ala')
>>> type (slownik.get('ala'))
<class 'NoneType'>
```

Możemy również sami wskazać wartość domyślną, załóżmy, że zależy nam na tym, aby była to wartość 0:

```
>>> slownik.get('ala',0)
0
```

W celu sprawdzenia czy dany klucz znajduje się w słowniku, można użyć składni `klucz in slownik`.

```
>>> 1 in slownik
False
```

```
>>> 'Ala' in slownik
True
```

Użyteczne są ponadto metody .**values**(), .**keys**(), .**items**(), które zwracają iteratory odpowiednio: wartości słownika, kluczy oraz par elementów klucz i wartość zwróconych jako krotki. Są to iteratory, więc umożliwiają odwoływanie się do poszczególnych elementów po kolei. Jednocześnie są to tzw. obiekty widoków, co oznacza, że są ściśle powiązane ze słownikami: zachowują kolejność elementów w słowniku oraz automatycznie odwzorowują zmiany w słowniku. Można je również np. przekształcić na listy:

```
>>> slownik.values()  # widok wartości
dict_values([1, 1, 0, 2])
>>> slownik.items() # widok elementów – krotek  klucz, wartość
dict_items([('Ala', 1), ('ma', 1), ('psa', 0), ('kota', 2)])
>>> slownik.keys() # widok kluczy
dict_keys(['Ala', 'ma', 'psa', 'kota'])
>>> for i in slownik.values():
...     print(i)
1
1
0
2
>>> list(slownik.keys())
['Ala', 'ma', 'psa', 'kota']
>>> widok = slownik.items()  #  widok elementów słownika
>>> widok
dict_items([('Ala', 1), ('ma', 1), ('psa', 0), ('kota', 2)])
>>> slownik['Ala'] =  0 # modyfikacja słownika – zmiana wartości
>>> del slownik['psa'] # modyfikacja słownika – usunięcie elementu
>>> widok
dict_items([('Ala', 0), ('ma', 1), ('kota', 2)])
```

Poniższy przykład przedstawia słownik służący przechowywaniu danych o transakcjach. Poszczególne klucze słownika to numer transakcji, kwota i waluta. Słowniki zawierające dane o poszczególnych transakcjach są umieszczone w zmiennej `lista_transakcji`. Załóżmy, że naszym zadaniem jest wstępne oczyszczenie danych polegające na usunięciu elementów, dla których nie podano numeru transakcji:

```
>>> transakcja_1 = {'numer': 1, 'kwota': 1000, 'waluta':
'PLN'}
>>> transakcja_2 = {'numer': 1001, 'kwota': 1000, 'waluta':
'PLN'}
```

```
>>> transakcja_3 = {'numer': 202, 'kwota': 2000, 'waluta':
'EUR'}
>>> transakcja_4 = {'numer': '', 'kwota': 0, 'waluta': ''}
>>> transakcja_5 = {'numer': 10101, 'kwota': 3500, 'waluta':
'USD'}
>>> transakcja_6 = {'numer': '', 'kwota': 0.5, 'waluta': ''}
>>> lista_transakcji = [transakcja_1, transakcja_2,
transakcja_3, transakcja_4, transakcja_5, transakcja_6]
>>> for transakcja in lista_transakcji:
...     if not transakcja.get('numer'):
...         lista_transakcji.remove(transakcja)
>>> lista_transakcji
[{'numer': 1, 'kwota': 1000, 'waluta': 'PLN'}, {'numer':
1001, 'kwota': 1000, 'waluta': 'PLN'}, {'numer': 202,
'kwota': 2000, 'waluta': 'EUR'}, {'numer': 10101, 'kwota':
3500, 'waluta': 'USD'}]
```

Posłużyliśmy się tutaj składnią `if not transakcja.get('numer')`. Wywołanie transakcja.**get**('numer') w przypadku braku wpisanego numeru zwraca `''` (pusty napis), który przyjmuje wartość logiczną *False*. A zatem w przypadku natrafienia na taki element, warunek if not transakcja.get('numer') sprawdza się do *if not False* czyli przyjmuje wartość *True*, co powoduje, że usuwamy element.

Podobnie jak dla list, tak dla słowników możliwe jest wykorzystanie słowników składanych (*dictionary comprehension*). Składnia jest analogiczna do list złożonych, z tą różnicą, że używamy nawiasów klamrowych {} zamiast kwadratowych [], a zatem jest to `{działanie for element in typ_iterowalny}`. Należy pamiętać że elementem słownika jest para klucz: wartość. Prześledźmy poniższe przykłady. Pierwszy z nich dla każdej litery w napisie zwraca parę litera: litera powtórzona 3 razy. Drugi natomiast dla liczb całkowitych z zakresu <0, 49> zwraca pary liczba: liczba podzielona przez 3 jeśli liczba jest (bez reszty) podzielna przez 3.

```
>>> {x : x * 3 for x in "abcd"}
{'a': 'aaa', 'b': 'bbb', 'c': 'ccc', 'd': 'ddd'}
>>> {x : x / 3 for x in range(50) if x % 3 == 0}
{0: 0.0, 3: 1.0, 6: 2.0, 9: 3.0, 12: 4.0, 15: 5.0, 18: 6.0,
21: 7.0, 24: 8.0, 27: 9.0, 30: 10.0, 33: 11.0, 36: 12.0,
39: 13.0, 42: 14.0, 45: 15.0, 48: 16.0}
```

2.6 Zbiory

Zbiór, jako typ w Pythonie, to kolekcja unikalnych i niezmienniczych elementów o nieustalonej kolejności. Zbiory zachowują podobieństwo do kluczy w słownikach. Do słowników podobne jest również tworzenie zbiorów, wykorzystujące nawiasy klamrowe.

```
>>> zbiór = {1, 3, 4, "kot"}
>>> zbiór
{1, 3, 4, 'kot'}
>>> type(zbiór)
<class 'set'>
```

Możemy również tworzyć zbiory używając polecenia **set** i elementu iterowalnego, np. napisu czy listy.

```
>>> set("abcd")
{'d', 'a', 'b', 'c'}
>>> set(["a","b","c","d"])
{'d', 'a', 'b', 'c'}
```

Składnia ta jest o tyle ważna, że pozwala na utworzenie pustego zbioru, ponieważ wywołanie {} tworzy pusty słownik.

```
>>> zbiór = set()
>>> type(zbiór)
<class 'set'>
```

W celu dodania elementu do zbioru używamy metody .**add**():

```
>>> zbiór.add("bb")
>>> zbiór
{'bb'}
```

Do usuwania elementów służą: .**pop**() — metoda wywoływana bez argumentu, która usuwa i zwraca element zbioru, ale nie mamy kontroli nad tym, który element to będzie oraz metoda .**remove**(element) — wywoływana z elementem zbioru w charakterze argumentu, który chcemy usunąć.

```
>>> zbiór = set("abecadło")
>>> zbiór
{'ł', 'c', 'o', 'b', 'd', 'e', 'a'}
>>> zbiór.pop()
'ł'
>>> zbiór
{'c', 'o', 'b', 'd', 'e', 'a'}
>>> zbiór.remove("a")
>>> zbiór
{'c', 'o', 'b', 'd', 'e'}
```

Inną metodą do usuwania elementów zbioru jest .**discard**(element). Działa ona w ten sposób, że jeśli usuwany element należy do zbioru, usunie go tak, jak metoda .**remove**(element). Jeśli zaś usuwany element nie należy do zbioru, wówczas nie zwróci błędu w przeciwieństwie do metody .**remove**(element):

```
>>> zbiór
{'o', 'b', 'd', 'e'}
>>> zbiór.discard('m')
>>> zbiór
{'o', 'b', 'd', 'e'}
>>> zbiór.remove('m')
Traceback (most recent call last):
  File "<pyshell#134>", line 1, in <module>
    zbiór.remove('m')
KeyError: 'm'
```

Działania na zbiorach obejmują: test przynależności dla elementów (operator *in*), sumę logiczną (operator | lub metoda .**union**()), iloczyn logiczny (operator & lub metoda .**intersection**()), różnicę zbiorów (operator -), zawieranie zbiorów (operatory <> lub metody .**issubset**(), .**issuperset**()), xor (operator ^).

```
>>> zbiór1 = {1, 2, 3}
>>> zbiór2 = {2, 3, 4}
>>> 1 in zbiór1
True
>>> 1 in zbiór2
False
>>> zbiór1 & zbiór2
{2, 3}
>>> zbiór1.intersection(zbiór2)
{2, 3}
>>> zbiór1 | zbiór2
{1, 2, 3, 4}
>>> zbiór1.union(zbiór2)
{1, 2, 3, 4}
>>> zbiór1 - zbiór2
{1}
>>> zbiór1 < zbiór2
False
>>> zbiór2 > zbiór1
False
>>> zbiór3 = {3}
>>> zbiór3 < zbiór1
True
>>> zbiór3.issubset(zbiór1)
True
>>> zbiór1.issuperset(zbiór3)
True
>>> zbiór1 ^ zbiór2
{1, 4}
```

Podobnie jak dla list i słowników, tak dla zbiorów możliwe jest wykorzystanie zbiorów składanych (*set comprehension*). Używamy nawiasów klamrowych {} zamiast kwadratowych , a zatem jest to `{działanie for element in`

typ_iterowalny}. Przykłady wykorzystania tej składni pomijamy, zachęcając Czytelnika w razie potrzeby do lektury np. Lutz (2011, s. 259–260).

Dodatkowe informacje dotyczące korzystania z pomocy: wywołanie funkcji **dir()** z argumentem wskazującym nazwę lub egzemplarz/obiekt danej klasy zwróci listę dostępnych atrybutów i metod, np. **dir**(str) wypisze atrybuty i metody dostępne dla typu *str*. Ten sam rezultat otrzymamy przy wywołaniu **dir**("abcd"):

```
['__add__', '__class__', '__contains__', '__delattr__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__ge-
tattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__
ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__
str__', '__subclasshook__', 'capitalize', 'casefold', 'cen-
ter', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'is-
decimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 're-
place', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Jeśli znamy nazwę polecenia (metody), ale nie jesteśmy pewni co do jego składni czy działania, możemy wywołać je używając polecenia **help**(egzemplarz_klasy.nazwa_metody), np.:

```
>>> help("abcd".isalpha)
Help on built-in function isalpha:

isalpha(...) method of builtins.str instance
    S.isalpha() -> bool

    Return True if all characters in S are alphabetic
    and there is at least one character in S, False other-
wise.
```

Pomoc uzyskamy również dzięki IDE (*Integrated Development Environment*), czyli zintegrowanemu środowisku programistycznemu, tj. narzędziu (programowi), który służy do pisania własnych programów, a którego funkcjonalność oprócz edycji tekstu (kodu) może polegać na przykład na „podpowiadaniu" metod, wskazywaniu typu zmiennych, podpowiedziach dotyczących składni itp.

Dodatkowe informacje dotyczące uczenia się na błędach: dobrym pomysłem jest nie tylko powtarzanie przykładów zawartych w niniejszej książce, lecz rów-

nież własna ich modyfikacja. Pozwoli to na utrwalenie składni oraz na naukę na własnych błędach. W tym miejscu zachęcamy Czytelników do zapoznawania się i oswajania z komunikatami błędów zgłaszanych przez Pythona. Przykładowo wywołanie:

```
>>> "abcd"[5]
```

zakończy się błędem, ponieważ napis napis ma 4 litery, więc próbujemy odwołać się do nieistniejącego znaku. Otrzymujemy następujący komunikat, informujący o przekroczeniu zakresu:

```
Traceback (most recent call last):
  File "<pyshell#115>", line 1, in <module>
    "abcd"[5]
IndexError: string index out of range
```

Podobnie na przykład próba dzielenia przez zero skończy się niepowodzeniem, o czym również dowiemy się z komunikatu o błędzie:

```
>>> 2 / 0
Traceback (most recent call last):
  File "<pyshell#116>", line 1, in <module>
    2/0
ZeroDivisionError: division by zero
```

Uważna lektura komunikatów Pythona zaoszczędzi Czytelnikowi sporo czasu (i stresu) w nauce programowania.

Dodatkowe informacje dotyczące innych typów liczbowych: do reprezentacji liczb wymiernych w Pythonie służy dedykowany typ *Fraction*. Zachowuje on zarówno licznik, jak i mianownik ułamka, aby ograniczyć niedogodności wynikające z reprezentacji liczb zmiennoprzecinkowych. Dokonamy najpierw importu odpowiedniego modułu, a następnie utworzymy ułamek używając składni `Fraction(licznik, mianownik)`. Przykład poniżej dotyczy ułamka 1/3.

```
>>> from fractions import Fraction
>>> Fraction(1, 3)
Fraction(1, 3)
```

Po utworzeniu odpowiednich obiektów, możemy przeprowadzić obliczenia z ich wykorzystaniem. W poniższym przykładzie dokonamy obliczenia tego samego wyrażenia z wykorzystaniem typu *Fraction* oraz typu *float* i sprawdzimy następnie czy wynik da spodziewany rezultat.

```
>>> d = Fraction(5, 10) + Fraction(2, 10) + Fraction(1, 10)
– Fraction(8,10)  # d = 0,5 + 0,2 + 0,1 - 0,8
>>> d  # zmienna powinna przechowywać 0
Fraction(0, 1)
```

```
>>> float(Fraction(0, 1))
0.0
>>> d = 0.5 + 0.2 + 0.1 - 0.8
>>> d
-1.1102230246251565e-16
```

2.7 Jaki typ danych wybrać?

Wybór typu przechowującego konkretne wartości niekiedy jest automatyczny (np. przechowywanie pojedynczych wartości typu *int* czy *float*), natomiast trudniejsze jest podjęcie decyzji dotyczących typów „pojemnikowych", np. listy, krotki czy słownika. Przy podjęciu decyzji należy zwrócić uwagę przede wszystkim na dwa aspekty funkcjonalne: możliwość modyfikacji oraz zachowanie kolejności danych. Na przykład jeśli wahamy się między listą a krotką, a nie jest konieczna modyfikacja danych, lepiej użyć tej drugiej (krotka działa szybciej od listy). Krotka będzie użyteczna również w sytuacji, kiedy nie będziemy chcieli dopuścić do zmiany przechowywanych danych. Z kolei w sytuacji, kiedy nie będziemy chcieli dopuścić do duplikowania wartości, wybierzemy zbiór, który nie pozwoli na powtarzanie swoich elementów. Co więcej, dla zbiorów operator `in` działa szybciej niż w przypadku list. Jednak w sytuacji, kiedy znaczenie dla nas będzie miała kolejność przechowywanych danych, zrezygnujemy ze zbioru na rzecz listy/krotki. Jak wiemy, w krotce i liście poszczególne elementy oznaczane są kolejnymi liczbami całkowitymi, a w słowniku stosuje się nazwy pól. Słownik może być wygodniejszy, kiedy nasze dane zawierają „różne" informacje — np. imię, nazwisko i kod pocztowy. Nie musimy wówczas pamiętać o ich kolejności.

Co ciekawe, słowniki szybciej obsługują operacje wstawiania i usuwania elementu (zob. Jaworski, Ziadé, 2017, s. 58, 64) — oczywiście kosztem tego, że słownik nie obsługuje wszystkich możliwości listy. Z drugiej strony, jeśli przetwarzamy spory zbiór danych i zależy nam na wydajności, wygodniejszą i szybszą opcją jest użycie modułu *pandas*, który za chwilę omówimy (rozdz. 2.8).

2.8 Moduł pandas

Moduł *pandas* oferuje wszechstronne i bardzo wygodne środowisko obsługujące dane w postaci tabel. Obsługuje on zarówno przekształcenia danych jak i podstawowe obliczenia.

Pandas nie jest modułem wbudowanym, dlatego powinien być zainstalowany (zob. rozdz. 1.4), a następnie każdorazowo zaczytywany (importowany). Decydując się na użycie *pandas*, będziemy używać go do większości operacji na danych, a więc dość często wywoływać jego funkcję i klasy. Dlatego na oznaczenie *pandas* będziemy stosować zwyczajowo przyjęty skrót (alias) — *pd*.

Dodatkowo powinniśmy zaimportować moduł *numpy* (aliasowany jako *np*) służący do obliczeń na wektorach i macierzach (jeśli tego nie zrobimy, moduł *pandas* i tak będzie korzystał z *numpy*, jednak niektóre wszystkie możliwości *pandas* nie będą wykorzystane). W takim polecenia importu wyglądają następująco:

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

Moduł *pandas* nie operuje na wbudowanych typach (strukturach) danych, np. listach czy słownikach, ale na własnych klasach — typach danych. Najprostszym typem danych jest *pd.Series* — jest odpowiednikiem pojedynczego szeregu statystycznego albo inaczej mówiąc — jednej kolumny w arkuszu kalkulacyjnym. Zmienna typu *pd.Series* przechowuje wartości *N* elementów (niekoniecznie liczbowych) o porządku określonym według indeksu obserwacji. Konstrukcja *pd.Series* wydaje się podobna do listy, jednak o ile lista może zawierać elementy różnego typu, to *Series* zawiera elementy jednego typu. Dzięki temu działania na obiektach *pd.Series* są lepiej zaprojektowane pod kątem szybkości i wygody przetwarzania danych. Szereg może zostać zainicjalizowany jako pusty albo można podać wartości z innego obiektu — np. listy. Ilustruje to poniższy kod.

```
>>> pustyszereg = pd.Series()
>>> szeregliczbowy = pd.Series([1, 2, 4, 7, 11])
```

Dla porządku dodamy, że zamiast listy można użyć także np. krotki, wektora *numpy* czy sekwencji *range*. W dalszej części nauczymy się importować szeregi z danymi ze zbioru .csv, poza tym możliwe jest np. pozyskanie z zapytania do bazy danych.

Charakterystyczne, że podstawowe działania na dwu szeregach działają element-po-elemencie, czyli dodawanie dwu szeregów spowoduje wygenerowanie nowego szeregu, którego elementy są sumą odpowiednich elementów (oczywiście w takim przypadku oba szeregi muszą mieć jednakową liczebność).

```
>>> a = pd.Series([1, 5, 7])
>>> b = pd.Series([2, 5, 7])
>>> print(a + b)
0     3
1    10
2    14
dtype: int64
>>> print(a / b)
0    0.5
1    1.0
2    1.0
dtype: float64
```

Podobnie jak przy dzieleniu *int* otrzymujemy *float*, tak dzielenie szeregów z liczbami całkowitymi zwróciło wynik w postaci szeregu *float*.

Pokreślimy w tym miejscu, że tylko dla dodawania i odejmowania operacje te są równoważne dodawaniu i odejmowaniu wektorów. Począwszy od Pythona 3.5 w pakiecie *pandas* (a także *numpy*) wprowadzono nowy operator: @ ('at'), dzięki czemu możliwe jest bezpośrednie mnożenie element po elemencie (*) i mnożenie wektorowe (@)[3]. Dla tych, którzy w przyszłości zamierzają używać pakietu *numpy* wspomnimy o tym, że *pandas* wykonuje operacje na odpowiadających sobie elementach według indeksu, a *numpy* indeksuje automatycznie od zera (tak jak lista), dlatego operuje na odpowiadających elementach wg kolejności.

Równie prosto możemy wykonać operację każdego elementu i pojedynczej wartości (skalaru). Np.

```
>>> pd.Series([1, 5, 10]) + 3
0     4
1     8
2    13
dtype: int64
>>> pd.Series([1, 5, 10]) * 3
0     3
1    15
2    30
dtype: int64
```

Także działanie operatorów relacyjnych (zob. rozdz. 1.3) jest odmienne niż dla dotychczas poznanych typów. Nie zwracają one jednej wartości *True*/*False*, ale wynik porównania element po elemencie. Rezultatem jest więc odpowiedniej długości szereg wartości logicznych. Przykładowo:

```
>>> pd.Series([1, 2, 5]) == pd.Series([1, 4, 10])
0     True
1    False
2    False
dtype: bool
>>> pd.Series([1, 2, 10]) > pd.Series([1, 4, 5])
0    False
1    False
2     True
dtype: bool
```

Podane podejście jest bardzo wygodnym sposobem porównania dwóch odpowiadających par liczb. Przykładowo mamy dwa szeregi zawierające wynagrodzenia w grudniu i styczniu poprzedniego roku. Przy pomocy podanych wyżej metod możemy stworzyć flagę (przyjmującą wartości *True*/*False*) wska-

[3] We wcześniejszych wersjach Pythona takie mnożenie było też możliwe, wymagało skorzystania z metody .**dot**() w module numpy (*np.dot(wektor1, wektor2)*).

zującą kto z pracowników nie dostał podwyżki (przykład z operatorem ==) albo kto taką podwyżkę dostał (przykład z operatorem >).

Podczas wykonywania obliczeń często potrzebujemy wyznaczyć zbiorcze charakterystyki liczbowe. Takie proste charakterystyki jak suma, średnia, odchylenie standardowe czy wartość najmniejsza i największa mogą być bezpośrednio policzone przy pomocy metod wbudowanych w obiekty *pd.Series*.

Tab. 2.1 Podstawowe metody wyznaczające charakterystyki liczbowe obiektu pd.Series (przyjmijmy, o nazwie: szereg1)

Operatory	Charakterystyka szeregu
szereg1.sum()	Suma elementów
szereg1.mean()	Średnia
szereg1.min(), szereg1.max()	Wartość najmniejsza i największa
szereg1.count()	Liczba niepustych elementów (nie zawsze równa liczbie wierszy)
szereg1.describe()	Zwraca osiem podstawowych charakterystyk (klasyczne i pozycyjne)

W praktyce przetwarzania danych częste zastosowanie mają metody pozwalające zwrócić informacje o zbiorze unikalnych wartości: szereg1.**unique**() lub liczbie unikalnych wartości: szereg1.**nunique**(). Pierwsza z wymienionych metod zwraca wektor, w którym każda z wartości występujących w szeregu źródłowym występuje wyłącznie raz, a druga — zwraca liczbę całkowitą równą długości wektora wartości unikalnych.

```
>>> szereg1 = pd.Series([0, 1, 0, 1, 2, 2, 0])
>>> szereg1.unique()
array([0, 1, 2])
>>> szereg1.nunique()
3
```

Co w przypadku, gdy chcielibyśmy obejrzeć nie tylko zbiór wartości występujących w szeregu, ale także podać liczbę elementów o danej wartości (tabulacja). Przykładowo, dla szeregu który identyfikuje płeć respondenta ('M', 'K'):

```
>>> plec = pd.Series(['M', 'M', 'K', 'K', 'M', 'K'])
>>> plec.value_counts()
M    3
K    3
dtype: int64
```

W powyższym przykładzie szereg 'plec' zawierał 3 obserwacje 'M' (mężczyzna) i 3 'K' (kobieta), taki też wynik otrzymano stosując **value_counts**(). Jeśli potrzebujemy użyć tych wartości, to pamiętajmy, że metoda **value_counts**() zwraca liczbę wystąpień w postaci obiektu *pd.Series* (indeks — to unikalne wartości oryginalnego szeregu).

W przetwarzaniu danych bardzo ważną własnością obiektów klasy *pd.Series* jest możliwość ich **indeksowania**. Pierwszy sposób indeksowania umożliwia odniesienie się do numeru obserwacji (indeksu). Poznaliśmy już przy okazji list. O ile w obiektach typu lista wartości indeksu oznaczają kolejność elementów w liście 'Lista' i przyjmują wartości od 0 do (len(Lista) -1), to w obiektach *pd.Series* można przyjąć inne wartości indeksu. Indeks, dostępny jest za pomocą metody .**index**, może więc oznaczać np. rok którego dotyczy dana obserwacja, numer transakcji, identyfikator klienta, itp.

W przypadku *pd.Series* możliwe jest także indeksowanie przy pomocy wartości logicznych. Konwencja ta polega na tym, że dla szeregu o długości *N*, zamiast podania numeru indeksu interesujących nas obserwacji, podajemy szereg *N* wartości logicznych. W rezultacie Python zwraca te wartości, dla których wartość logiczna była *True*. Jak zobaczymy w dalszych przykładach, indeksowanie boolean jest bardzo wygodne, gdy chcemy wybrać elementy spełniające dany warunek.

Poniższy przykład ilustruje wybór dwu pierwszych elementów, kolejno według standardowej konwencji (wykorzystujących indeks obserwacji) i boolean.

```
>>> szereg = pd.Series([0, 2, 4, 6])
>>> szereg[0:2]
0    0
1    2
dtype: int64
>>> szereg[[True, True, False, False]]
0    0
1    2
dtype: int64
```

Wielu Czytelników zapewne zada sobie pytanie: dlaczego w powyższym przykładzie podawaliśmy cztery wartości boolean, podczas gdy wystarczyło podać interesujący nas zakres? Oczywiście, gdy wiemy o które (w sensie indeksu) obserwacje chodzi, wówczas wygodniej jest korzystać z indeksowania według numeru indeksu. W praktyce szeregi mogą liczyć tysiące lub miliony obserwacji. W takiej sytuacji zdecydowanie lepiej sprawdzi się podanie warunku i indeksowanie boolean, zamiast ręcznego (czasochłonnego i podatnego na ludzkie błędy) wyszukiwania wartości spełniających określony warunek. Przykład takiego indeksowania jest podany w kolejnym akapicie. Z drugiej strony, indeks nie zawsze odpowiada numerowi obserwacji (np. możemy indeskować pracowników po numerze PESEL i w takim przypadku pierwsza osoba w naszym zbiorze danych na pewno nie będzie mieć indeksu 0). Jak wprowadzić ,,niestandardowy'' (czyli inny niż domyślny *zero-indexing*) indeks? Wystarczy w trakcie tworzenia *pd.Series* podać wartości w parametrze 'index'. Przykładowo:

```
>>> pd.Series([0, -5, 102.5], index=[7, 77, 88])
7       0.0
77     -5.0
88    102.5
dtype: float64
```

Wróćmy do poleceń pozwalających na wybór poszukiwanych elementów szeregu. Indeksowanie przy pomocy wartości logicznych jest naturalnym sposobem wyboru z szeregu tych obserwacji, które spełniają zadany warunek. Na przykład, gdy interesuje nas wybór elementów większych od 10, wówczas kod przedstawiałby się następująco:

```
>>> szereg = pd.Series([0, 15, 9, 16])
>>> szereg[szereg > 10]
1    15
3    16
dtype: int64
```

Dla upewnienie się, że wiemy jak dokładnie działa powyższa operacja, zobaczmy jak wygląda wynik zadanego wyrażenia logicznego.

```
>>> print(szereg > 10)
0    False
1     True
2    False
3     True
dtype: bool
```

W podobny sposób możemy oczywiście wykonać operacje na parze szeregów o jednakowej długości (np. sprawdzając warunek `szereg1 > szereg2`).

Dodatkowo, szereg zawierający wartości boolowskie (logiczne) może sam być przedmiotem dalszych operacji. Oprócz standardowych operatorów logicznych (np. ORAZ, LUB), warto zwrócić na **.all()** i **.any()**. Pierwszy z nich (all) sprawdzają one czy wszystkie elementy są *True*, a drugi (any) czy choć jeden element jest *True*. Przykłady poniżej ilustrują te działania.

```
>>> pd.Series([True, True, False, False, True]).any()
True
>>> pd.Series([True, True, False, False, True]).all()
False

>>> szereg = pd.Series([0, 15, 9, 16])
>>> (szereg > 0).all()
False
>>> (szereg > 0).any()
True
```

Rzadko kiedy przeprowadzamy operacje jednym szeregu lub kilku osobnych szeregach. Znacznie częściej przechowujemy dane w taki sposób, że ten sam indeks obserwacji jest przypisany do wielu szeregów. Taką konstrukcję ma arkusz kalkulacyjny, ale także tabela SQL. W module *pandas* takim obiektem jest ramka danych (pd.DataFrame)[4].

Elementy (kolumny) dataframe mają te same własności do pd.Series. Poprzez analogię do arkuszy kalkulacyjnych (np. Excel) możemy powiedzieć, że *pd.Series* odpowiada pojedynczej kolumnie, ale *pd.DataFrame* — całemu arkuszowi. Oczywiście taka analogia daje wyłącznie podstawowy pogląd, gdyż ramka danych z biblioteki *pandas* posiada dużo szersze możliwości niż arkusze kalkulacyjne (i nieporównanie lepszą wydajność przy przetwarzaniu dużych zbiorów danych).

„Ręczne'' tworzenie nowego obiektu pd.DataFrame zwykle odbywa się na bazie słownika list (lub słownika innych typów sekwencyjnych — np. krotek lub range()).

```
>>> df1 = pd.DataFrame({"a": [1, 1], "b": [2, 9]})
>>> df1
   a  b
0  1  2
1  1  9
```

Działanie standardowych operatorów (+, -, *, /, %, **) oraz operatorów relacyjnych (np. ==, >) działa element elemencie. Wymaga się więc, żeby oba obiekty miały taką samą liczbę wierszy (tak jak dla *pd.Series*), a operacje wykonywane są na odpowiadających sobie kolumnach. W przypadku, gdy brakuje odpowiadających sobie kolumn wstawiane są wartości NaN (jest to wartość nieokreślona liczbowo albo oznaczająca brak obserwacji, pomijana w obliczeniach i koncepcyjnie zbliżona do 'nan' wspomnianego w rozdz. 2.1) albo zwracany jest błąd (np. dla operatorów relacyjnych).

```
>>> df1 = pd.DataFrame({"a": (0, 2, 4), "b": (1, 2, 3)})
>>> df2 = pd.DataFrame({"a": (0, 3, 4), "b": (0, 0, 0)})
>>> df1 + df2
   a  b
0  0  1
1  5  2
2  8  3
>>> df2 = pd.DataFrame({"a": (0, 3, 4), "c": (0, 0, 0)})
>>> df1 + df2
   a   b   c
0  0 NaN NaN
1  5 NaN NaN
2  8 NaN NaN
```

[4] Podobne ramki danych (DataFrame lub data.table) są bardzo popularne w pakiecie R (zob. Biecek, 2014 oraz Kopczewska i in., 2016).

Oczywiście nic nie stoi na przeszkodzie, aby stworzyć ramkę danych z wcześniej utworzonych szeregów (obiektów *pd.Series*). Przykładowa składnia to `pd.DataFrame({„a”: szereg_a, „b”: szereg_b})`. Taki sposób będzie bardzo naturalny, gdy pozyskujemy poszczególne szeregi z różnych źródeł lub przetwarzamy dane.

Działanie operatorów `df1 [operator] liczba` bądź: `liczba [operator] df1` pozwala wykonać odpowiednie działanie (także wyniki operatora relacyjnego). Co istotne, operacja wykonywana jest dla wszystkich elementów data-frame (czyli wykonywana jest po wierszach i kolumnach). Zapis taki jest bardzo czytelny i elegancki w zapisie, a jest doskonałym przykładem dobrego stylu (*Pythonic*). Spójrzmy na krótkie przykłady.

```
>>> df1 = pd.DataFrame({"a": (0, 2, 4), "b": (1, 2, 3)})
>>> df1 * 10
    a   b
0   0  10
1  20  20
2  40  30
>>> 1 + df1
   a  b
0  1  2
1  3  3
2  5  4
```

Obiekt typu pd.DataFrame posiada kilka interesujących pól, które przechowują jego „własności''. Oto najczęściej stosowane:

Tab. 2.2 Wybrane atrybuty i metody obiektu pd.DataFrame (przyjmijmy, o nazwie: df1)

Operatory	Zawiera
df1.size	Liczbę obserwacji wchodzącej w skład ramki danych (liczba wierszy * liczba kolumn)
df1.shape	Wymiary ramki danych (zwykle dwa, odpowiednio: liczba wierszy i kolumn)
df1.columns	Nazwy kolumn (przechowywane w obiekcie pandas, podobnym do pd.Series)
df.head()	Nagłówek ramki (nazwy zmiennych i kilka początkowych wartości każdej zmiennej)
df.tail()	Końcówka ramki (nazwy zmiennych i kilka ostatnich wartości każdej zmiennej)
df.sample(n)	Wybierz losowo *n* wierszy z ramki

Czytelnicy mogą sprawdzić działanie samodzielnie, co do czego zachęcamy. Dodamy tylko, że większość operacji przedstawionych z Tab. 2.2 można zastosować nie tylko do ramki, ale i do pojedynczego szeregu.

Pandas oferuje wiele możliwości i zapewne większość prostych i nieco bardziej skomplikowanych metod przetwarzania danych i dalszych analiz jest już zaimplementowana w *pandas* (zaczynając od prostej sumy, średniej itp.). Dla-

tego, o ile to jest możliwe, do standardowych operacji lepiej poszukać i użyć wbudowanych metod. Oszczędza to czas programisty (nie trzeba na nowo „wymyślać koła'') i obniża prawdopodobieństwo popełnienia błędu (znacznie łatwiej jest popełnić błąd pisząc własną funkcję niż korzystając z funkcji sprawdzonej przez zespół twórców danej biblioteki i liczne środowisko jej użytkowników). W przypadku *pandas* dodatkowym argumentem jest czas. Metody „wbudowane'' działają wektorowo, a co za tym idzie — znacznie szybciej niż pętle napisane w Pythonie, wykonujące iteracje np. na tysiącach wierszy.

Wiele tzw. działań agregujących — np. sumowanie, minimum, maksimum — może być wykonane „w pionie'' (w rezultacie otrzymamy wynik tego działania dla każdej kolumny) lub „w poziomie'' (wtedy otrzymamy wynik działania dla każdego wiersza). Zachęcamy Czytelnika do wypróbowania poleceń podanych niżej na uprzednio zdefiniowanej ramce *df1*.

– suma kolumn DataFrame: *df1.sum(axis=0)*;
– suma wierszy DataFrame: *df1.sum(axis=1)*;
– analogicznie możemy policzyć średnie (**.mean()** zamiast **.sum()**), odchylenie standardowe (**.std()**), wartość największą/najmniejszą (**.min()**, **.max()**).

W podanym wyżej przypadku takim prostszym rozwiązaniem jest pomnożenie ramki danych przez skalar (co skutkuje pomnożeniem każdego z jej elementów przez skalar, podobnie jak w przypadku macierzy).

```
>>> print(df1 * 3)
   a   b
0  3   6
1  3  27
```

Nasze programy powinny odczytywać dane źródłowe i przechowywać je. Wielu użytkowników przechowuje dane i wyniki obliczeń w plikach arkusza kalkulacyjnego (np. Excel). Wspominaliśmy już zresztą analogię ramki danych z arkuszem Excel. *Pandas* obsługuje pliki arkuszy Excel, jednak sam Excel zmienia co kilka lat standard zapisu pliku. Ponadto musieliśmy odwoływać się nie tylko do pliku, ale i nazwy konkretnego arkusza. Dlatego będziemy namawiać Czytelników na odczytywanie i zapis danych przy pomocy plików *.csv* (a jeśli zajdzie taka potrzeba, w bardzo podobny sposób możemy obsłużyć pliki Excela i inne; zob. przykład podany w załączniku). Dwie najważniejsze metody do obsługi takich plików to *pd.read_csv()* oraz *pd.to_csv()* (odpowiednio do odczytu i zapisu ramki danych do pliku *.csv*).

Wspominane metody zawierają wiele parametrów opcjonalnych. Najważniejsze z nich to 'sep' (znak oddzielający komórki w poziomie — domyślnie jest to przecinek, ale gdy stosujemy przecinek jako separator liczb dziesiętnych, możemy zastosować inny znak, np. średnik lub tabulator) i 'decimal' (separator liczb dziesiętnych — domyślnie jest to kropka, według konwencji krajów anglosaskich, ale możemy używać notacji polskiej i zapisywać ułamki z uży-

ciem przecinków). Jak zawsze zachęcamy Czytelnika do lektury dokumentacji w celu zapoznania się z innymi parametrami.

Wspominaliśmy już wycinki z obiektu *pd.series*. Podobne operacje (stosując zarówno indeksowanie boolean jak i indeksy pozycyjne) możemy przeprowadzić nie tylko na pojedynczych kolumnach ramki danych, ale także na całej ramce danych:

```
>>> df1 = pd.DataFrame({"a": (0, 1, 22), "b": (-1, -5, 9)})
>>> df1[[True, True, False]]
   a  b
0  0 -1
1  1 -5
```

Ciekawostka: jedną z ciekawszych metod w ramach *pd.DataFrame* jest sekwencja **.iterrows**(). Dzięki niej możemy użyć pętli po wierszach. Sekwencja **.iterrows**() zwraca dwie wartości (krotkę): indeks i zawartość wiersza.

```
>>> for iindex, irow in df1.iterrows():
...     print(irow * 3)
...
a    3
b    6
Name: 0, dtype: int64
a     3
b    27
Name: 1, dtype: int64
```

Ciekawostka: ciekawą operacją jest filtrowanie kolumn. Możemy w ten sposób „wybrać" ramkę danych zawierają wyłącznie kolumny nas interesujące, bądź odwrotnie — wykluczyć pewne kolumny z ramki danych.

```
df.filter(like='a')
```

Załącznik: łączenie zawartości plików Excela

Wyobraźmy sobie, że posiadamy informacje zgromadzone w wielu (np. kilka tysięcy) plikach Excela. Na szczęście pliki te umieszczono w jednym katalogu, co więcej — nie ma tam innych plików z rozszerzeniem *.xls*. Chcielibyśmy zgromadzić zawartość wszystkich plików w jednym pliku i kolejność dołączania każdego z nich nie ma znaczenia (np. zawsze możemy posortować końcowy plik).

Zamiast ręcznie przeklejać zawartość każdego pliku do nowego arkusza Excel[5], spróbujemy połączyć informacje z tych plików z wykorzystaniem *pandas*. Jak w wielu innych przykładach, możemy to zrobić na różne sposoby, ale jak zobaczymy — moduł *pandas* jest wygodny także w przypadku tak prostych operacji. Użyjemy także modułu *os*, który pozwoli nam wyszukać interesujące pliki w katalogu. Zakładamy, że Czytelnicy zaimportują te moduły przed sprawdzeniem podanych komend. Dla ułatwienia cały przykład podzieliliśmy na etapy.

Etap 1: zmiana katalogu i znalezienie nazw wszystkich plików:

```
>>> os.chdir(r'C:\Moje Dokumenty\Pliki_Excel')
>>> wszystkie_pliki = os.listdir()
```

Etap 2: z listy wszystkich plików wybieramy tylko te z rozszerzeniem .xls (tj. nazwa pliku kończy się '.xls'; wykorzystujemy jedną z metod dedykowanych operacjom na napisach — zob. rozdz. 4.1):

```
>>> xls_pliki = [i for i in wszystkie_pliki if i.endswith('.
xls')]
```

Uwaga: krok ten możemy pominąć, jeśli w katalogu znajdują się wyłącznie pliki XLS.

Etap 3: utworzenie pustej ramki danych — „magazynu" gdzie będą dołączane poszczególne pliki

```
>>> polaczone = pd.DataFrame()
```

Etap 4: pętla po wszystkich plikach: wczytanie zawartości pliku do ramki danych i dołączenie zawartości do naszej ramki 'polaczone'

```
>>> for inazwa in xls_pliki:
>>>     jeden_xls = pd.read_excel(inazwa)
>>>     polaczone = polaczone.append(jeden_xls)
```

Uwaga: zakładamy, że wszystkie dane, które łączymy znajdują się w pierwszym arkuszu. W przeciwnym razie powinniśmy podać nazwę arkusza w opcjonalnym parametrze 'sheet_name'.

[5] Przyjmijmy, że otwarcie i zamknięcie każdego pliku zajmuje sekundę, a operacje kopiuj–wklej każdego pliku *.xls* zajmie sprawnemu pracownikowi biurowemu tylko 5 sekund (bardzo optymistyczne założenie). Łącznie z czasem niezbędny na utworzenie nowego pliku, zapis pliku wynikowego i wyrywkowym sprawdzeniem wyników cała operacja dla 1000 plików wymaga około 2 godzin pracy. Natomiast czas potrzebny do automatycznego łączenia plików z użyciem *pandas* na przeciętnym komputerze biurowym nie przekraczał 15 sekund.

Etap 5: zapis wyniku do pliku ‘Wszystkie_pliki.xls’ — do arkusza ‘RAZEM’; dodatkowo pomijamy zapis numerów wierszy:

```
>>> polaczone.to_excel('Wszystkie_pliki.xls', sheet_
name="RAZEM", index=False)
```

3. Maszynowe pobieranie danych ze stron i serwisów internetowych

Popularne stwierdzenie mówi, że ,,wszystko jest w internecie''. Oczywiście jest w tym nieco przesady, ale zasoby internetu są rozległym źródłem informacji. Choć trudno oszacować wielkość zasobów internetu, to przykładowe szacunki mówią o 3 miliardach stron internetowych i ponad 4 miliardach użytkowników (https://www.websitehostingrating.com/internet-statistics-facts), gdzie obie podane wielkości corocznie powiększają się o 5–10%.

Okazuje się, że Python jest bardzo dobrym narzędziem do pobierania i przetwarzania treści umieszczonych w internecie. W rozdziale dowiemy się więc jak zautomatyzować pobieranie danych ze stron internetowych, a także z serwisów przy pomocy interfejsu — tzw. API (to ostatnie odbywa się poprzez wysłanie do serwisu prośby o odpowiednio przygotowane dane a następnie ich pobranie).

W ostatnim rozdziale przedstawimy wyniki analizy danych zgromadzonych w serwisie ogłoszeń motoryzacyjnych.

Automatyczne (maszynowe) pobieranie zawartości stron internetowych i przekształcenie treści tych stron do postaci ,,ustrukturyzowanej'' (np. arkusza kalkulacyjnego lub plików tekstowych z osobno zapisanymi poszczególnymi elementami strony) nazywane jest z angielskiego *web scraping* (przegląd np. Edelman, 2012). Przykładowe zastosowania technik *web scraping* obejmują:

– pobieranie cen ze sklepów internetowych (Macias, Stelmasiak, 2019);
– pobieranie informacji o filmach i innych wydarzeniach kulturalnych, w tym recenzji, czasu i miejsca wydarzenia (Verma, Verma, 2019);
– pobieranie informacji związanych ze sportem np. informacje o meczach (termin, frekwencja, wynik) lub o zawodnikach;

– analiza treści ogłoszeń nieruchomości (Beręsewicz, 2016), ogłoszeń o pracę (Dusi i in., 2015; Maślankowski, 2019) lub aukcji internetowych (Lucking–Reiley et al., 2007, Baranowski, Komor, Wójcik, 2018).

3.1 Podstawowe metody pobierania danych ze strony internetowej

Zastanówmy się co właściwie robi przeglądarka (nasz program do pobierania zawartości strony będzie robił to samo) w celu wyświetlenia strony internetowej. Nie wchodząc nadmiernie w szczegóły techniczne, pobieranie strony internetowej wymaga ustanowienia połączenia ze serwerem o podanym adresie URL (np. www.time.org) i wysłania do niego żądania pobrania określonej zawartości (HTTP GET). Jeśli serwer zaakceptuje i prawidłowo przetworzy nasze żądanie, zwróci kod odpowiedzi 200 i prześle nam odpowiednią treść. W przypadku błędu zwróci najczęściej kody: 404 (podany adres nie istnieje), 403 (serwer zrozumiał nasze żądanie, ale nie udzielił nam dostępu do żądanego elementu) lub 500 (błąd po stronie serwera). Pełną listę kodów odpowiedzi wraz opisem znajdziemy np. na stronie https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

W Pythonie wysłanie żądania pobrania strony i pobranie jej zawartości można wykonać za pomocą wbudowanego modułu *urllib* lub modułu *requests* (który należy pobrać i zainstalować standardowo — przy pomocy 'pip'). Polecamy *requests*, który lepiej obsługuje kodowanie UTF-8 i pozwala łatwiej przekształcać wynik do postaci bezpośrednio obsługiwanej przez Python (np. słowniki). Zyskujemy także więcej możliwości „formatowania'' adresu, z czego będziemy korzystać przy odwoływaniu się do interfejsów (zob. rozdz. 3.5).

Początkowo nawiążemy połączenie przy pomocy polecenia requests.**get**(). Metoda .**get**() wymaga podania prawidłowego adresu strony lub innego obiektu (np. pliku). Ważne przy tym, żeby podać pełny adres, łącznie z nazwą protokołu (w przypadku stron internetowych najczęściej 'http://' lub 'https://'). Następnie korzystając z tego połączenia pobierzemy zawartość strony do zmiennej `kod_html_strony`. Poniżej przedstawiamy przykład dla strony http://pogodynka.pl.

```
>>> import requests
>>> strona = requests.get('http://www.pogodynka.pl/')
>>> kod_html_strony = strona.text
```

Działanie modułów *urllib* i *requests* nie ogranicza się do zawartości stron internetowych. Równie dobrze możemy wpisać adres pliku, który chcemy pobrać (np. zdjęcie twórcy języka Python https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/Guido-portrait-2014.jpg/360px-Guido-portrait-2014.jpg).

Na potrzeby technik *web scraping*, należy wprowadzić rozróżnienie na strony statyczne i dynamiczne. Rozróżnienie to jest istotne dla wyboru modułu, którym będziemy się posługiwać do automatycznego pozyskiwania informacji ze stron. Wszystkie strony statyczne mogą być obsłużone modułem *BeautifulSoup*, natomiast strony dynamiczne mogą wymagać użycia bardziej złożonego modułu *selenium*.

Strony statyczne posiadają treść zapisaną wprost za pomocą języka HTML. Zawartość strony statycznej jest niezmienna, tzn. wyświetla się jednakowo w każdym czasie i dla każdego użytkownika. Strony statyczne posiadają bardzo prostą konstrukcję, ale posiadają wiele ograniczeń i dlatego stanowią mniejszość.

Strona dynamiczna posiada zawartość każdorazowo generowaną za pomocą aplikacji znajdującej się na serwerze (zresztą takie aplikacje coraz częściej są pisane w Pythonie, zob. frameworki webowe wymienione w rozdz. 1.9). Taka aplikacja zmienia pokazywaną treść strony w zależności od zachowania użytkownika albo kontekstu (pory dnia, lokalizacji komputera z którego łączymy się z serwerem, zawartości ciasteczek). Statyczny kod HTML jest używany do stron zawierających niewiele treści albo treści, które rzadko się zmieniają. Może to być strona z podstawowymi informacjami o firmie czy instytucji albo strony poświęcone jednemu tematowi, prowadzone „hobbystycznie". Większe serwisy (portale społecznościowe, blogi, rozbudowane sklepy internetowe) zwykle będą dynamiczne.

W kolejnym podrozdziale nauczymy się przetwarzać kod HTML, tak aby „wydobyć" z niego potrzebne informacje.

3.2 Przetwarzanie kodu HTML

Strony internetowe napisane są w języku HTML (HyperText Markup Language). Język HTML oparty jest o znaczniki. Znaczniki są to wyrażenia ujęte w nawiasy trójkątne. Najczęściej znacznik jest używany dwukrotnie — pierwszy raz wskazuje na początek (znacznik otwierający, np. pogrubienie:), a drugi raz wskazuje na koniec fragmentu oznaczanego znacznikiem (znacznik zamykający, np. koniec pogrubienia). Znacznik zamykający jest tworzony poprzez powtórzenie znacznika otwierającego poprzedzonego znakiem „/" przed nazwą znacznika. Tak więc kod HTML

```
Tekst <b>pogrubiony</b> oraz <i>kursywa</i>
```

wygeneruje na stronie zawartość, która wygląda mniej więcej tak:

Tekst **pogrubiony** oraz *kursywa*.

Typowy dokument HTML zawiera nagłówki (head) i treść (body), a wszystko zawarte jest wewnątrz znaczników: otwierającego <html> i zamykającego </html>.

```
<html>
<head>
  <title>Tytul dokumentu HTML</title>
</head>
<body>
<p>To jest pierwszy akapit.</p>
<p>Drugi akapit: tekst <b>pogrubiony</b> oraz <i>kursywa</
i>.</p>
</body>
</html>
```

Przeszukiwanie zawartości stron wymaga podstawowe znajomości podstaw HTML, dzięki temu wiemy czego szukamy. Tab. 3.1 prezentuje najczęściej spotykane znaczniki używane do budowy stron HTML. Przykłady zaprezentowane dalej nie będą wykraczały poza znaczniki zaprezentowane w Tab. 3.1 (szersza prezentacja HTML — zob. np. Schafer, 2013, rozdz. 1, 3 i 4).

Tab. 3.1 Podstawowe znaczniki HTML

Znacznik otwierający	Znacznik zamykający	Znaczenie
<html>	</html>	Początek i koniec dokumentu HTML
<body>	</body>	Główna treść dokumentu
<h1>	</h1>	Nagłówek pierwszego poziomu (można stosować różne poziomy, od <h1> do <h6>)
		Krój czcionki: pogrubiony
		Krój czcionki: pochyły
		Początek i koniec akapitu
 Tekst odnośnika		Odnośnik do strony 'www.uni.lodz.pl', wyświetlony na stronie jako Tekst odnośnika
		Lista nienumerowana (wypunktowanie)
		Lista numerowana
		Element listy (występuje wewnątrz listy numerowanej lub nienumerowanej, np. pomiędzy znacznikami a)
		Blok
<table>	</table>	Tabela
<tr>	</tr>	Wiersz tabeli (występuje wewnątrz tabeli, pomiędzy znacznikami <table> a </table>)

Znacznik otwierający	Znacznik zamykający	Znaczenie
<td>	</td>	Kolumna tabeli (występuje wewnątrz tabeli, pomiędzy znacznikami <tr> a </tr>)
 	brak	Złamanie linii (nowy wers)

Przetwarzania strony należy zacząć od pobieżnego oglądu i próby „powiązania" zawartości widocznej w przeglądarce z kodem HTML. Obecnie jest to tyle łatwe, że w popularnych przeglądarkach (Firefox, Chrome) możemy najechać kursorem na danych element i po kliknięciu prawym klawiszem wybrać „zbadaj element".

Rys. 3.1 Widok w „inspektorze kodu" (Firefox)

Parsowanie (czyli przetwarzanie do postaci zawierającej logiczną strukturę) kodu HTML może odbywać się np. za pomocą *BeautifulSoup*[1]. Kontynuując przykład przetwarzania strony pogodynka.pl możemy znaleźć pierwszą tabelę (zawiera największe miasta) oraz wyodrębnić wiersze z tej tabeli.

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(kod_html_strony, 'html.parser')
>>> tabela_1 = soup.find('table')
>>> lista_wierszy = tabela_1.find_all('tr')
```

[1] *BeautifulSoup* to zwyczajowo używana nazwa, a Python używa nazwy *bs4*. Moduł ten nie jest wbudowany i powinien być wcześniej zainstalowany (zob. rozdz. 1.4).

Spójrzmy po kolei na przedstawiony kod. Zmienna `soup` przechowuje całą stronę, zawiera przetworzoną zawartość strony, m.in. podstawowe informacje o stronie (tytuł, osobno nagłówki i komentarze) i co najważniejsze — strukturę strony oraz metody ułatwiające nawigację po stronie:

– .**find**(znacznik) — metoda ta znajduje blok strony wyznaczony przez otwarcie i zamknięcie danego znacznika,
– .**find_all**(znacznik) — metoda ta zwraca listę znalezionych elementów HTML i jest wykorzystywana niemal w każdym programie do przetwarzania stron.

Uwaga: **find**() i **find_all**() przyjmują także wzorzec poszukiwania w postaci wyrażeń regularnych (zob. rozdz. 4.2).

Zmienna `tabela_1` jest egzemplarzem typu (klasy) 'bs4.element'. Także `lista_wierszy`, będąca wynikiem tego programu będzie składała się z obiektów typu 'bs4.element'. Jest to typ specjalnie zaprojektowany do obsługi stron HTML, dlatego odwołując się do atrybutów i metod możemy przeprowadzać wiele potrzebnych operacji:

– Zwrócenie tekstu wyświetlanego na stronie przez dany element (np. `lista_wierszy[0].text`);
– Zwrócenie elementu nadrzędnego — np. dla elementów `lista_wierszy` będzie to tabela (prosimy Czytelnika o samodzielne wyświetlenie wyrażenia `lista_wierszy[0].parent is tabela_1`) bądź podrzędnego (.**child**; w podanym przykładzie wiersze nie posiadają elementów podrzędnych, ale są podrzędne w stosunku to tabeli, a tabela jest podrzędna do bloku pomiędzy znacznikami i); w przypadku gdy poszukujemy wszystkich elementów podrzędnych możemy użyć iterowalnego obiektu .**children**;
– Załóżmy, że szukamy nie tylko pośrednie elementy podrzędne, ale także elementy podrzędne elementów podrzędnych (czyli w notacji „rodzinnej" nie tylko dzieci, ale wszyscy potomkowie). Wszystkie te elementy zawiera artybut .**descendants**;
– Wyszukanie wewnątrz (`tabela1.find_all()` — tę operację już przeprowadziliśmy).

Spójrzmy na bardziej rozbudowany przykład dotyczący ogłoszeń sprzedaży samochodów w serwisie gratka.pl (https://gratka.pl/motoryzacja/osobowe). W rozdziale skoncentrujemy się na pobraniu danych, a w Załączniku zaprezentujemy wyniki prostego badania statystycznego.

Interesować nas będzie przebieg, rok produkcji oraz pojemność silnika.

```
<ul class="teaser__params">
    <li>Przebieg: 81588</li>
    <li>Rodzaj ogłoszenia: sprzedaż</li>
    <li>Typ nadwozia: inny</li>
    <li>Rok produkcji: 2016</li>
```

```
    <li>Rodzaj paliwa: benzyna + gaz</li>
    <li>Pojemność silnika [cm3]: 1368</li>
</ul>
```

Standardowo musimy pobrać stronę i zapisać ją w obiekcie typu *BeautifulSoup*. Poznaliśmy już wszystkie niezbędne narzędzia i biblioteki, dlatego ten fragment kodu zamieszczamy bez omówienia.

```
>>> import requests
>>> from bs4 import BeautifulSoup
>>> strona = requests.get('https://gratka.pl/motoryzacja/
osobowe')
>>> kod_html_strony = strona.text
>>> soup = BeautifulSoup(kod_html_strony, 'html.parser')
```

Kolejnym krokiem będzie znalezienie wszystkich ogłoszeń. Przeglądając kod HTML strony łatwo zauważyć, że znalezienie wszystkich znaczników 'ul' zwróci również elementy inne niż treść ogłoszeń. Dlatego prosta metoda .**find_all**('ul') nie jest w tym przypadku najlepszym rozwiązaniem. Zamiast tego użyjemy .**find_all**() z dodatkowym argumentem, który określa atrybuty znacznika (czyli precyzuje, że szukamy tylko tych elementów, które będą zawierały atrybut 'class' o wartości 'teaser__params'). Widzieliśmy w kodzie HTML dotyczącym ogłoszenia, że każdy element-ogłoszenie zawiera elementy potomne. Dlatego listę elementów-ogłoszeń (w zmiennej `ogloszenia`) przetworzymy dalej. Wewnątrz każdego ogłoszenia (pierwszy poziom pętli for) będziemy poszukiwać znaczników 'li'. Spośród wszystkich tych znaczników do listy zebranych danych (`dane_przebieg`) dodamy jedynie te, które zawierają tekst zaczynający się od 'Przebieg' (drugi poziom pętli for w połączeniu z warunkiem if).

```
>>> ogloszenia = soup.findAll('ul', {'class': 'teaser__pa-
rams'})
>>> dane_przebieg = []
>>> for i in ogloszenia:
...     for j in i.find_all('li'):
...         if j.text.startswith('Przebieg'):
...             dane_przebieg.append(j.text)
...
```

Zanim przejdziemy do omówienia wyniku wygenerowanego przez podany kod, zauważmy że przetworzyliśmy tylko jedną stronę, zawierającą kilkadziesiąt ogłoszeń. Taką operację moglibyśmy równie dobrze wykonać bez użycia programu napisanego w Pythonie. Jeśli chcielibyśmy zgromadzić dane o wszystkich ogłoszeniach publikowanych na dany moment, należy powtó-

rzyć podany wyżej kod dla kolejnych stron — wywołując następną stronę[2]. Element wskazujący na następną stronę, widoczny jako „daszek skierowany w prawo'' (zbliżony do znaku większości: >). Dokonując „inspekcji'' przy pomocy przeglądarki widzimy, że element ten kryje się w znaczniku 'a' o atrybucie `class="pagination__nextPage"`.

W efekcie dane_przebieg zawiera treść (tekst) tych podpunktów, które odnoszą się do przebiegu. Dane zawarte w liście mają postać napisów np. [‚Przebieg: 24540', ‚Przebieg: 123000'], a więc nie mają postaci liczbowej, którą łatwo przetworzymy. Szczęśliwie dla nas, wszystkie te podpunkty zaczynają się od napisu 'Przebieg: '. Jak widać początek, który jest nam zbędny ma stałą długość, dlatego możemy skorzystać z wycinku napisu (rozdz. 2.2)[3].

```
>>> dane_przebiegi = [i[10:] for i in przebiegi]
```

Oczywiście podobnie jak w większości programów *web scraping*, ekstrakcję danych ze strony można rozwiązać na kilka sposobów. Przykładowo, moglibyśmy zauważyć, że przebieg i inne najważniejsze informacje na temat ogłoszenia znajdują się w kolumnie o parametrze 'id' równym 'rightColumn'. Z kolei w tej kolumnie przebieg znajduje się w trzecim z kolei znaczniku <li>. Innym rozwiązaniem (nie zalecanym, bo najbardziej podatnym na błędy!) byłoby wyszukanie słowa przebieg i znalezienie pierwszej liczby następującej po tym słowie.

Podobny warunek — `j.text.startswith('Rok produkcji:')` pozwoli na pobranie roku produkcji (oczywiście dane te gromadzilibyśmy w innej liście, np. 'dane_rokprodukcji').

Z kolei poniższy kod pozwoli na pobranie ceny. Sama zasada działania kodu niczym nie różni się od pobierania przebiegu i roku produkcji, stąd pominiemy jego dokładne omówienie.

```
>>> ogloszenia2 = soup.find_all('p', {'class': 'teaser__pri-
ce'})
>>> ceny = [i.text for i in ogloszenia2]
```

W praktyce opisany powyżej kod działa poprawnie jedynie w przypadku gdy każde ogłoszenie posiada informację o przebiegu i roku produkcji. W przypadku gdy znacznika <li> z przebiegiem nie ma (np. ogłoszenie dotyczy samochodu

[2] Alternatywą dla dodania zewnętrznej pętli (wówczas mielibyśmy pętle o trzech poziomach, co jest mało eleganckie i w bardziej rozbudowanych programach może skomplikować kod na tyle, że łatwiej o błędy) może być przepisanie części kodu poszukującego przebieg do funkcji. W książce nie omawiamy funkcji, ale Czytelnik znajdzie wprowadzenie do funkcji w Pythonie w innych podręcznikach.

[3] Wydaje się, że lepszym rozwiązaniem byłoby skorzystanie z wyrażeń regularnych (rozdz. 4.2). Dzięki temu program będzie odporny na odstępstwa w długości „nieliczbowego'' początku. Takie odstępstwa mogą wystąpić np. gdy na stronie pojawi się więcej niż jedna spacja po fragmencie "Przebieg:" albo pominięty zostanie znak dwukropka.

fabrycznie nowego albo sprzedawca nie wprowadził przebiegu) — informacja nie zostanie dodana do listy. W ten sposób otrzymamy listy parametrów (przebieg, rok produkcji i cena) o różnej długości. Utrudni do analizę zależności pomiędzy różnymi cechami (parametrami ogłoszenia), np. gdy chcemy policzyć korelację pomiędzy przebiegiem a rokiem produkcji (rozdz. 2, Załącznik). (przykładowo: wyobraźmy sobie, że w piątym ogłoszeniu brakuje roku produkcji; w takim przypadku piąty element listy `dane_rok` będzie zawierał przebieg zebrany z szóstego ogłoszenia; co gorsze, kiedy używamy zwykłej listy, nie będzie to w żaden sposób zaznaczone).

Dlatego zmodyfikujemy nieco kod, tak aby program zbierał informacje o brakach danych[4]. Dla każdego ogłoszenia przypiszemy początkowo zmienne 'i_przebieg' oraz 'i_rok' równe -99.

```
>>> ogloszenia = soup.findAll('ul', {'class': 'teaser__pa-
rams'})
>>> dane_przebieg = []
>>> dane_rok = []
>>> for i in ogloszenia:
...     i_przebieg = -99
...     i_rok = -99
...     for j in i.find_all('li'):
...         if j.text.startswith('Przebieg'):
...             i_przebieg = j.text
...         if j.text.startswith('Rok produkcji'):
...             i_rok = j.text
...     dane_przebieg.append(i_przebieg)
...     dane_rok.append(i_rok)
...
```

Bardzo często informacje, które chcemy pozyskać ze stron, zawarte są w tabelach (znacznik '<table>'). Takie fragmenty kodu HTML możemy przekształcić do znanego nam typu *pd.DataFrame* (rozdz. 2.8) przy pomocy funkcji pd.**read_html**(). Wynikiem działania tej funkcji jest lista zawierająca obiekty DataFrame. Czytelnika zachęcamy do samodzielnego wypróbowania działania tego polecenia na przykładowej tabeli HTML.

3.3 Strony dynamiczne. Moduł selenium

Moduł *BeautifulSoup* obsługuje strony statyczne oraz niektóre strony dynamiczne. Jednak możemy spotkać strony np. w technologii AJAX gdzie kod HTML — który normalnie widzimy w przeglądarce jest generowany przez interpreter przeglądarki, a nie przez serwer. W takim przypadku *requests* nie przechwyci pełnej zawartości strony. Nie oznacza to, że takiej strony nie można

[4] Zwyczajowo w badaniach społecznych, dla zmiennych co do których spodziewamy się liczby nieujemnej stosuje się liczbę -99 jako informację o braku odpowiedzi.

przetwarzać w Pythonie. Rozwiązaniem jest np. skorzystanie z modułu *selenium*. Obsługuje on strony internetowe poprzez wywołanie przeglądarki i oferuje dostęp do elementów już przetworzonych przez tę przeglądarkę. Nie ma więc znaczenia w jakiej technologii napisana jest strona, gdyż przetwarzamy treść strony która wyświetla się w przeglądarce. *Selenium* oferuje również pełną nawigację po stronie — z łatwością możemy więc „klikać" w znalezione elementy, przestawiać menu rozwijane czy przyciski, wpisywać tekst do formularzy a nawet wykonywać czynności takie jak przewinięcie strony.

Web scraping za pomoca *selenium* wymaga nie tylko instalacji modułu 'selenium', ale także użycia „sterownika" stanowiącego połączenie *selenium* z naszą przeglądarką (https://docs.seleniumhq.org/download). W chwili pisania tej książki istniały sterowniki do popularnych przeglądarek: Mozilla Firefox (Mozilla GeckoDriver) oraz Chrome (Google Chrome Driver). Sterownik *selenium* do oferowanego przez Microsoft Edge był wydany dopiero w chwili kończenia tej książki (czerwiec 2020). Dlatego nasza wiedza o połączeniu *selenium* z Edge jest bardzo ograniczona i zainteresowani Czytelnicy powinni wypróbować ten sterownik na własną rękę.

Skopiowany sterownik należy umieścić na dysku, najlepiej w katalogu, który jest w ścieżce dostępu albo dodamy go do ścieżki dostępu. Jeśli z jakiegoś powodu nie mamy dostępu do zmiennej środowiskowej PATH, zapamiętajmy ten katalog i podamy go bezpośrednio w kodzie naszego programu (taki przypadek pokażemy w przykładzie).

Przykłady poniżej będą prezentowane dla Firefox, jednak niewielka zmiana początkowych ustawień pozwoli korzystać ze sterownika dla Chrome.

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.by import By
>>> path_driver = r'C:\my\path\geckodriver.exe'
>>> driver = webdriver.Firefox(executable_path=path_driver)
>>> driver = webdriver.Firefox(firefox_binary="firefox")
```

Uwaga: wartość zmiennej `path_driver` jest podana w konwencji tzw. surowej notacji napisu, stąd napis poprzedzony jest literą 'r'. Szerzej opisujemy to w rozdz. 4.1, a teraz tylko uprzedzamy Czytelnika, że taki zapis jest celowy, a nie jest wynikiem np. błędu drukarskiego.

Po wstępnej konfiguracji możemy wywołać stronę — weźmy jako przykład serwis banku Millenium.

```
>>> driver.get('https://www.bankmillennium.pl/')
```

Poszukiwanie elementów, podobnie jak w przypadku *Beatiful Soup*, możemy przeprowadzić szukając jednego (driver.**find_element**()) lub wielu elementów (driver.**find_elements**()). Wynik działania metody **find_element**() możemy przypisać do zmiennej (będzie to obiekt typu WebElements, jest to specyficzny typ dla modułu *selenium*). Natomiast **find_elements**() zwróci listę składającą

się z takich obiektów. Ponieważ selenium oferuje znacznie więcej możliwości, metody te wymagają podania na podstawie jakich własności chcemy wyszukać elementy. Dysponujemy m.in. możliwością wyszukania według znacznika (By.TAG_NAME), nazwy klasy (By.CLASS_NAME), id elementu (By.ID) ale także bardziej zaawansowanych: według ścieżki Xpath (By.XPATH) oraz selektora stylu CSS (By.CSS_SELECTOR). Sama składnia zapytania nie zmienia się, ale do użycia tych zapytań niezbędna jest wiedza na temat ścieżki Xpath oraz definicji stylów CSS. Zainteresowani Czytelnicy mogą znaleźć więcej szczegółów w książce Duckett (2015, rozdz. 5).

Co możemy zrobić ze znalezionymi elementami:

– kliknąć w element (.**click**());
– pobrać tekst wyświetlany przez dany element (.text);
– wyszukać „wewnątrz" danego elementu (podobnie jak w przykładzie *Beatiful Soup* szukaliśmy znacznika 'ul' i wewnątrz tego fragmenty HTML — znaczników 'li');
– wpisać tekst (.**send_keys**('Tekst wpisany do formularza')) i wysłać formularz (.**submit**()).

Czytelnik zapewne już zauważył, że operacja `driver.get()` otwiera przeglądarkę (a gdy ta wcześniej była otwarta — nowe okno) i tam wyświetla żądaną stronę. To dlatego selenium sprawia, że zawartość stron wczytywanych automatycznie nie różni się od tego co są otrzymalibyśmy ręcznie przeglądając daną stronę. Jest to także bardzo wygodne w procesie tworzenia i testowania programu, gdyż poszczególne operacje są widoczne.

Dla strony www.bankmillennium.com.pl spróbujmy przedstawić przykład z formularzem. Zakładamy, że wcześniej został zainicjalizowany *webdriver* i wywołaliśmy adres strony (przy pomocy `driver.get()`). Najpierw znajdziemy element zawierający pole formularza (np. po nazwie — na omawianej stronie banku jest to jedyny element o nazwie 'q' oraz jedyny o ID 'search'). W kolejnym kroku wpiszemy do formularza słowo 'python', a na końcu przekażemy zawartość pola do serwera. Zachęcamy Czytelnika do wywołania tych poleceń pojedynczo i obserwowania zachowania przeglądarki.

```
>>> szukaj_form = driver.find_element(By.NAME, 'q')
>>> # alternatywna wersja: driver.find_element(By.ID, 'se-
arch')
>>> szukaj_form.send_keys('kursy walut')
>>> szukaj_form.submit()
```

W trakcie nawigacji na stronie może pojawić się potrzeba pozyskania bieżącego adresu. Czynność tę przeprowadzimy za pomocą komendy `driver.current_url` (tak uzyskaną wartość możemy przypisać do zmiennej i np. zapisać do arkusza kalkulacyjnego bądź bazy danych). Kiedy zakończymy już przeglądanie strony, warto całkowicie zakończyć działanie programu — zamknąć przeglarkę. Służy temu komenda `driver.close()`.

3.4 Niezawodność scrapera. Obsługa wyjątków

Podczas automatycznego pobierania danych z internetu „coś może pójść nie tak". Po pierwsze, serwer obsługujący stronę może być chwilowo wyłączony albo przeciążony. Po drugie, nasz komputer może utracić połączenie z internetem. Po trzecie — jeżeli przeglądamy kilkaset stron, jest bardzo prawdopodobne, że któryś z podanych adresów zmieni się (np. ktoś wycofa ogłoszenie, które nie jest już aktualne).

W programach automatycznie pobierających dane powinniśmy przewidzieć te sytuacje i je odpowiednio „obsłużyć". „Obsłużyć" — oznacza, że nie powinno się dopuszczać, żeby typowy błąd przerwał działanie programu. Jest to ważne, kiedy program zbiera duże ilości danych i nie mamy możliwości „nadzoru" nad tym czy wszystkie operacje wykonują się prawidłowo (np. program zbiera dane 24 godziny na dobę; w takim przypadku w nocy „ręcznie" nie przywrócimy jego działania np. uruchomiając program na nowo lub modyfikując jego parametry). Zamiast przerywać działanie programu, w przypadku wystąpienia błędu można np. poczekać 15 sekund i ponowić próbę albo w przypadku gdy „scrapujemy" kilka serwisów — pominąć jeden i przejść do innej strony (dzięki temu stracimy dane z jednego serwisu, ale ciągle będziemy zbierali pozostałe). W przypadku, gdy strona nie istnieje, błąd przerwałby działanie całego programu. W wielu przypadkach — niepotrzebnie, bo jeśli brakuje tylko jednej strony (jeszcze raz odwołajmy się do przykładu nieaktualnych ogłoszeń), to przecież można kontynuować zbieranie informacji z pozostałych stron.

Generalnie obsługę takich zdarzeń możemy przeprowadzić dwojako. Możemy w programie wprowadzić instrukcje warunkowe (if, if-else), których celem będzie sprawdzenie czy spełnione są podstawowe wymogi, np.: podano właściwe adresy stron internetowych, kod odpowiedzi strony jest równy 200, itd. W ten sposób czynimy nasz program odporny na te błędy, które jesteśmy w stanie przewidzieć i dobrze zdefiniować ich źródła. Instrukcje warunkowe omówiliśmy już w rozdz. 1.4, dlatego przejdziemy do obsługi wyjątków.

Wyjątkiem nazywamy sytuację, w której program w sposób nieoczekiwany przerywa swoje działanie. Zwykle określamy to jako „błąd", jednak źródłem tego błędu nie jest błędna składnia polecenia (np. wpisanie komendy 'if' bez dwukropka na końcu linii[5]), ale okoliczności wykonania polecenia. Innymi słowy, możliwe, że w innych okolicznościach program wykona się poprawnie i nie przerwie swojego działania. Przykładowo:

– wpisanie 'a = 2/0' skutkuje wyjątkiem *ZeroDivisionError*, zbliżony koncepcyjnie wyjątek *ValueError* wystąpi w przypadku, gdy podajemy argument właściwego typu, ale dla danej wartości interpreter nie jest w stanie zwrócić wyniku (np. chcemy obliczyć pierwiastek kwadratowy z liczby ujemnej albo usunąć z listy element, który nie znajduje w zadanej liście);

[5] Omówienie najważniejszych przyczyn błędów składniowych i sposobów szybkiego wykrycia ich źródła w kodzie znajduje się w: Rother (2018, s. 35–37).

– odwołanie do nieistniejącej nazwy zmiennej skutkuje wyjątkiem *NameError*; podobnie w przypadku importu interpreter zgłosi wyjątek *ModuleNotFoundError*, gdy chcemy importować moduł, który nie jest zainstalowany (zob. rozdz. 1.5);
– w przypadku gdy lista zawiera dwa elementy, odwołanie się do trzeciego[6] i kolejnych elementów skutkuje wyjątkiem *IndexError*; podobny wyjątek (*KeyError*) zostanie zgłoszony gdy odwołujemy się do nie istniejącego klucza (oba te wyjątki są przypadkami nadrzędnej klasy wyjątków *LookupError*);
– wykonanie operacji nie obsługiwanej dla danego typu zmiennej (np. możenie listy przez liczbę niecałkowitą `[0, 5] * 1.5`, dodawanie i mnożenie słownika czy dodanie liczby i napisu) skutkuje wyjątkiem *TypeError*.

Obsługa wyjątków wymaga podania instrukcji, które obejmiemy „nadzorem" do osobnego bloku (`try`), którego granice wyznacza zastosowania wcięć w kodzie (podobnie jak w przypadku instrukcji warunkowych, pętli itp.). Następnie podajemy blok `except`, który zostanie wykonany w przypadku, gdy wystąpi wyjątek. Spójrzmy na przykład obsługi wyjątku, kiedy wprowadzimy wartości wydatków i liczbę osób, a następnie chcemy przeliczyć średnią wartość wydatków na osobę:

```
>>> wydatki = 100
>>> osob = 0
>>> try:
...     print(wydatki/osob)
... except:
...     print('Podaj liczbe osob wieksza od zera')
...
Podaj liczbe osob wieksza od zera
```

Powyższy kod zadziała zgodnie z oczekiwaniami w przypadku dzielenia przez zero. Jednak blok `except` zostanie wywołany także w przypadku każdego innego wyjątku (np. *NameError* w przypadku gdy zapomnimy wprowadzić jedną ze zmiennych — sprawdzenie pozostawiamy Czytelnikowi). Dlatego lepszą praktyką jest określenie nazwy wyjątku bezpośrednio za słowem kluczowym `except`. W takim przypadku możliwe jest podanie więcej niż jednego bloku `except`. Spójrzmy na przykład liczenia średniej wartości wydatków na osobę.

```
>>> try:
...     print(wydatki/osob)
... except ZeroDivisionError:
...     print('Podaj liczbe osob wieksza od zera')
... except NameError:
...     print('Brak zmiennej "wydatki" lub "osob"')
```

[6] Ten przypadek może także wynikać z prostego błędu programisty, np. niezastosowania zasady indeksowania listy od zera, gdyż indeks [2] listy oznacza trzeci a nie drugi element listy.

```
...
Podaj liczbe osob wieksza od zera
```

Ogólne wyjątki rzadko znajdą zastosowanie w programach do web scrapingu. Z doświadczeń autorów tej książki wynika, że najczęstsze problemy ze stroną dotyczą problemów z połączeniem lub zajętością serwera na którym umieszczono stronę. Przestawimy krótko najczęściej występujące wyjątki (pełną listę można znaleźć w dokumentacji lub wpisując `dir(requests.exceptions)`).

Tab. 3.2 Najczęściej spotykane wyjątki modułu *requests*

Wyjątek	Opis
requests.exceptions.ConnectTimeout requests.exceptions.Timeout	Przekroczono czas oczekiwania na nawiązanie połączenia z serwerem lub na odpowiedź serwera
requests.exceptions.ConnectionError	Brak połączenia z internetem, nie istnieje strona o podanym adresie, serwer odmawia połączenia
requests.exceptions.HTTPError	–
requests.exceptions.InvalidURL	Podano nieprawidłowy adres strony (np. konieczne jest oznaczenia protokołu sieciowego — dodanie prefiksu 'http://' lub 'https://')
requests.exceptions.RequestException	Ogólny wyjątek, oznacza bliżej nie sprecyzowany błąd połączenia

Uwaga: moduł *requests* standardowo nie zgłasza wyjątków, żeby obsłużyć wyjątki i korzystać z `except`: należy wewnątrz bloku try „ręcznie" włączyć wyjątki wywołując polecenie: **.raise_for_status()**. Przykładowy kod podsumuje nasze rozważania.

```
>>> try:
...     strona = requests.get('http://jakisadres')
...     strona.raise_for_status()
... except requests.exceptions.Timeout:
...     # spróbuj np. poczekać kilka sekund i wywołać po-
nownie
...     # w razie gdy i to nie pomoże – dopiero wyjatek
...     print('Zbyt dlugi czas oczekiwania')
... except requests.exceptions.ConnectionError:
...     print('Problem z polaczeniem ze strona')
...
Problem z polaczeniem ze strona
```

Wyjątki są również obsługiwane przez selenium. Ich pełną listę można znaleźć `selenium.common.exceptions`.

Tab. 3.3 Najczęściej spotykane wyjątki modułu *selenium*

Wyjątek	Opis
NoSuchElementException	Element do którego się odwołujemy nie istnieje
ElementNotVisibleException	Element do którego się odwołujemy nie jest widoczny na stronie, co uniemożliwia interakcję
InvalidSelectorException	Podano nieprawidłowy wzorzec wyszukania elementu (zwykle nieprawidłową składnię ścieżki Xpath)
TimeoutException	Przekroczono czas oczekiwania na nawiązanie połączenia z serwerem lub na odpowiedź serwera
WebDriverException	Błąd związany z uruchomieniem aplikacji webdriver, która kontroluje przeglądarkę

Na szczególne omówienie zasługuje wyjątek *StaleElementReferenceException*, który pojawia się gdy element który próbujemy odczytać (albo uruchomić) wcześniej istniał, ale nie jest już dostępny. Najczęściej powodem jest to, że strona zmieniła „wygląd", np. szukamy elementu na stronie głównej (np. ikony czy linka z menu głównego), choć w międzyczasie nasz program wcisnął przycisk „wyszukaj" i aktualnie widzimy wyniki tego wyszukiwania a menu główne jest niedostępne.

Wspomnijmy krótko o subtelnej różnicy pomiędzy *InvalidSelectorException* a *NoSuchElementException* (ewentualnie *ElementNotVisibleException*). W pierwszym przypadku problem wynika z tego, że samo polecenie wyszukiwania jest nieprawidłowe. Najczęściej problem wynika z nieprawidłowej składni polecenia Xpath, dlatego zaleca się sprawdzić:

- znaki cudzysłowu — np. Xpath `'//*[@class="button"]'` jest nieprawidłowe, zgodnie z zasadami Python dopuszcza się stosowanie pojedynczych cudzysłowów wewnątrz podwójnych bądź na odwrót, dlatego prawidłowe zapytanie będzie np. `'//*[@class="button"]';`
- sprawdzić czy nawiasy kwadratowe wewnątrz polecenia Xpath zostały otwarte a następnie zamknięte;
- dokładną nazwę polecenia, np. chcemy poszukać nagłówka pierwszego poziomu który zawiera tekst „podsumowania" (uwaga na wielkość liter!) i użyjemy polecenia `'//h1[contain(text(),"podsumowania")]'` podczas gdy właściwą nazwą funkcji jest 'contains' — prawidłowo powinno być: `'//h1[contains(text(),"podsumowania")]'`.

Natomiast w przypadku *NoSuchElementException* składnia polecenia Xpath jest prawidłowa, co oznacza że *selenium* jest w stanie rozpoznać czego szukamy, jednak poszukiwany element nie istnieje. Dobrym przykładem jest uruchomienie polecenia:

```
>>> szukaj_form = driver.find_element(By.NAME, 'q')
>>> szukaj_form.send_keys('kursy walut')
```

Bardzo często, jeśli nie zawsze, w programach napotykamy na problem zbyt szybkiego pobierania danych ze strony internetowej. Po prostu czas, w jakim komputer wykonuje polecenie (np. przejście do kolejnej podstrony) jest tak szybki, że połączenie internetowe (lub serwer) nie nadąża przesłać danych do naszego programu. W rezultacie nasz program zgłosi wyjątek Timeout (lub inny podobnie brzmiący). Inną kwestią jest to, że serwer może nie dopuścić do wielu, szybko następujących po sobie połączeń z jednego komputera, co stanowi zabezpieczenie przed niektórymi cyberatakami (DoS lub D-DoS, zob. np. Gierszewski, Molisz, 2014).

Standardową receptą na te problemy jest spowolnienie pracy, np. dodanie w programie komendy time.**sleep**(*ilesekund*) (pamiętajmy zaimportować moduł *time*). Starajmy się przewidzieć na jakich etapach pobierania zawartości strony wymagany jest czas na „odpoczynek", dzięki czemu nasz scraper nie działa zbyt szybko. W *selenium* mamy jeszcze inną możliwość, czyli skorzystanie z jednego z dwu sposobów wymuszenia czasu oczekiwania. Pierwszy sposób to użycie metody .**implicitly_wait**(*ilesekund*), czyli ustawienie globalnego limitu czasu oczekiwania zaraz po inicjacji 'webdriver'. Dla przykładu podanego w rozdz. 3.3 wyglądałoby to tak:

```
>>> driver.implicitly_wait(10)
```

Domyślna wartość tego parametru to 0,5 sekundy, co w wielu przypadkach jest wartością niewystarczającą. W praktyce lepiej zwiększyć ten czas przynajmniej do 10 sekund, szczególnie, że jest maksymalny limit czasu wykonania — czyli jeśli czynność zostanie wykonana szybciej niż przewiduje to nasz limit, to program nie będzie oczekiwał aż do upływu 10 sekund, tylko od razu przejdzie do następnych komend. Drugi, rzadziej stosowany, sposób to *explicit wait*, czyli zadeklarowanie czasu oczekiwania na pojedyncze zdarzenie (którym może być np. wczytanie współrzędnych elementu lub jego widoczność — pojawienie się na stronie, dostępność przycisku — możliwość kliknięcia itp.). Kiedy stosujemy *explicit wait*? Krótko mówiąc gdy nie potrzebujemy ustawiać globalnie długiego limitu czasu oczekiwania albo celowo dla pojedynczej czynności chcemy ten czas zmniejszyć (np. do testów).

Kod dla strony www.bankmillennium.pl (tym razem spróbujmy aktywować przycisk logowania, który znajduje się po kliknięciu elementu oznaczonego znacznikiem '<a class="bmp-btn bmp-btn--blue modal-trigger">'; dla przypomnienia i ułatwienia zrozumienia przykładu podajemy cały kod, wraz z importem modułów i wywołaniem strony).

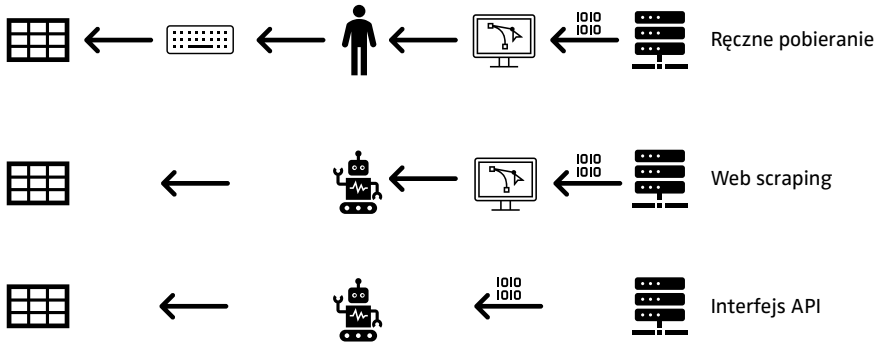
```
>>> from selenium import webdriver
>>> from selenium.webdriver.support.ui import WebDriverWait
>>> from selenium.webdriver.support import expected_con-
ditions
>>> from selenium.webdriver.common.by import By
>>> path_driver = r'C:\my\path\geckodriver.exe'
```

```
>>> driver = webdriver.Firefox(executable_path=path_driver)
>>> driver = webdriver.Firefox(firefox_binary="firefox")
>>> driver.get('https://www.bankmillennium.com.pl/')
>>> element = WebDriverWait(driver, 5).until(expected_con-
ditions.element_to_be_clickable((By.XPATH, "//a[@class='bm-
p-btn bmp-btn--blue modal-trigger"']")))
>>> element.click()
```

3.5 API

Nie zawsze musimy używać *web scraper* w celu pozyskania danych przez internet. Niektóre strony oferują gotowe usługi udostępniania lub przetwarzania danych przy pomocy tzw. API (*application programming interface*). API jest interfejsem, czyli zestandaryzowanym sposobem przekazywania danych maszyna-maszyna. Ponieważ interfejs ten służy do przekazywania danych pomiędzy różnymi aplikacjami, nie posiada warstwy graficznej i innych rzeczy, które ułatwiają percepcję człowieka (w przeciwieństwie do klasycznych serwisów internetowych). Porównanie trzech sposobów zbierania danych z internetu przedstawia Rys. 3.2.

Rys. 3.2 Zbieranie danych z internetu przez człowieka, web scraper oraz API



Schemat przedstawiony na Rys. 3.2 pokazuje, że ręczne pobieranie danych przez człowiek wymaga przesłania danych z serwera do przeglądarki (znajdującej się na komputerze lokalnym), a następnie człowiek przetwarza te informacje np. przepisuje lub ręcznie wkleja dane z odpowiednich fragmentów strony do arkusza kalkulacyjnego. W przypadku użycia techniki *web scraping* nadal dane są przesyłane w postaci stron HTML, ale zamiast człowieka przetwarza je skrypt lub program. W przypadku skorzystania z interfejsu API program pozyskuje bezpośrednio niezbędne informacje z serwera i zupełnie można pominąć warstwę graficzną (HTML).

Gdy API jest dostępne, niemal zawsze jest to rozwiązanie preferowane wobec *web scraping*[7]:

– Serwis może zastrzec w regulaminie (*term of use*), że korzystanie z niego przez roboty (programy komputerowe) jest zabronione (szerzej: Mitchell, 2015, dodatek C);
– Podczas *web scraping* mogą wystąpić błędy, których nie obsłużymy prawidłowo; pobranie przez API jest bardziej bezpośrednie i prawdopodobieństwo wystąpienia błędu jest mniejsze;
– Strony zmieniają swoją zawartość, niekiedy na tyle znacząco, że nasz web scraper przestanie prawidłowo działać; API zmieniają się rzadziej a w przypadku zmian użytkownicy są uprzedzeni znacznie wcześniej.

Wśród licznych przykładów oferowanych interfejsów API można też wymienić interfejsy oferowane przez banki. Interfejsy te są umożliwiają klientom i osobom przez nich uprawnionym dostęp m.in. do historii operacji bankowych. Na mocy dyrektywy PSD2, od września 2019 roku udostępnienie takich danych za pomocą zestandaryzowanych interfejsów stało się obowiązkiem banków (zob. Brener, 2019; Paduszyńska, Pawlak, 2020).

Większość API wymaga uzyskania dostępu (np. jest płatne, przeznaczone dla zamkniętej grupy — np. klientów danej firmy albo służy kontrolowaniu limitu użycia serwisu przydzielonego danemu użytkownikowi). Najczęściej serwis oferujący API, po otrzymaniu wniosku o dostęp (np. wypełnienie formularza rejestracyjnego albo zalogowanie się do serwisu, którego jesteśmy klientami) przekazuje klucz — czyli unikalny kod dostępu. Następnie użytkownik korzystający z API podaje ten klucz przy każdym zapytaniu.

Zaczniemy od przykładu API Narodowego Banku Polskiego, za pomocą którego można pobierać kursy walutowe. Interfejs API NBP nie wymaga podania klucza dostępu, ponadto jego użycie jest bardzo proste i polega jedynie na wywołaniu odpowiedniego adresu. Przykładowo, aby pozyskać kurs funta (GBP) z dnia 30.04.2018 wystarczy wywołanie w przeglądarce adresu „http://api.nbp.pl/api/exchangerates/rates/a/gbp/2018-04-30". Końcówka (sufiks) adresu ('gbp' i '2018-04-30') zmienia się więc w zależności od treści zapytania. Na skutek takiego wywołania serwer NBP zwróci następujący dokument (w formacie XML, zbliżonym do HTML):

```
<ExchangeRatesSeries>
<Table>A</Table>
<Currency>funt szterling</Currency>
<Code>GBP</Code>
<Rates>
```

[7] Wspomina o tym Jarmin (2019, s. 174) — wskazując, że urzędy statystyczne eksperymenujące z wykorzystaniem danych gromadzonych przy pomocy web scrapingu, starają się pozyskiwać te dane we współpracy z firmami (sklepami). Po nawiązaniu takiej współpracy głównym sposobem technicznej obsługi wymiany danych staje się korzystanie z interfejsów API.

```
  <Rate><No>084/A/NBP/2018</No>
  <EffectiveDate>2018-04-30</EffectiveDate>
  <Mid>4.7888</Mid>
  </Rate>
</Rates>
</ExchangeRatesSeries>
```

Możemy więc bez trudu użyć znanej już metody .**get**() z modułu *requests*, używając odpowiednich operacji na łańcuchach znaków (napisach).

```
>>> import requests
>>> adres_baz = 'http://api.nbp.pl/api/exchangerates/ra-
tes/a/'
>>> adres_pelny = adres_baz + 'gbp/' + '2018-04-30'
>>> strona = requests.get(adres_pelny)
>>> strona_zawartosc = strona.text
```

Tak otrzymana zmienna `strona_zawartosc` zawiera ciąg znaków. Jeśli nie chcemy przeszukiwać zawartości tekstu, możemy przekształcić zawartość do słownika za pomocą modułu *requests*[8]. Dzięki temu zachowujemy strukturę wynikowego dokumentu (sposób ten jest bardzo przydatny w przypadku bardziej złożonych wyrażeń).

```
>>> wynik_w_slowniku = strona.json()
```

Znacznie częściej interfejsy API obsługują zapytania przy pomocy „parametrów" podanych bezpośrednio po adresie strony — serwisu obsługującego API. Standardowo parametry są przekazywane poprzez podanie po znaku zapytania par klucz-wartość (analogicznie jak w słowniku). Na przykład dla hipotetycznego API adres.api.pl adres generujący zapytanie dla parametrów *klucz1* o wartości 'ABCD' i *klucz2* o wartości 250 byłby następujący:

```
adres.api.pl?klucz1=ABCD&klucz2=250
```

Patrząc na powyższy przykład zwróćmy uwagę, że poszczególne pary klucz-wartość są oddzielone znakiem tzw. ampersand (&). Żeby uświadomić sobie jak bardzo popularny jest taki sposób przekazywania zapytań do serwisu internetowego wystarczy wpisać jakiekolwiek słowo (wyrażenie) do wyszukiwarki internetowej (np. *google.com*, *bing.com* czy *duckduckgo.com*). Po zatwierdzeniu pojawi się adres zawierający pary klucz-wartość oddzielone znakiem ampersand.

Konstrukcja zapytania zawierającego znak zapytania i pary oddzielone ampersandem nie jest trudna i takie zapytanie może być stworzone za pomocą prostych operacji na napisach — zob. rozdz. 4. Jest to jednak mozolne, poza

[8] Inną możliwością jest użycie modułu *json*, a ściślej: funkcji json.**loads**(). JSON (JavaScript Object Notation) to standard umożliwiający zapis obiektów podobnych do słownika (par klucz + wartość) w formacie tekstowym.

tym problemy napotkamy gdy chcemy przekazać wartości zawierające spacje i inne „białe znaki''. Dlatego do generowania takich zapytań wystarczy wykorzystać dodatkowe możliwości request.**get**().

Dlatego bezpośrednie wywołanie adresu 'adres.api.pl?klucz1=ABCD&klucz2=250' możemy zastąpić następująco:

```
>>> requests.get('adres.api.pl', params={'klucz1': 'ABCD',
'klucz2': '250'})
```

Jak wspomniano, większość API wymaga uzyskania dostępu. Hasła te również mogą być podane w postaci parametrów (**Uwaga:** dokumentacja serwisów API często hasła nazywa kluczami; nie należy tego mylić z kluczami w sensie słownika, bo w rzeczywistości hasła te powinny podawane w słowniku jako wartości!). Dobrym zwyczajem jest przechowywanie haseł poza kodem źródłowym, np. w osobnym pliku. Dzięki temu przekazując kod źródłowy innym osobom unikniemy przypadkowego podania parametrów dostępu do naszego konta w serwisie API.

W niektórych przypadkach (np. API serwisu Twitter czy linii lotniczej Lufthansa) spotyka się bardziej złożony sposób autoryzacji. W pierwszej kolejności wysyła się dane do serwera (np. przy pomocy komendy request.**post**) z podanymi danymi autoryzacyjnymi. Serwer wysyła odpowiedź z tymczasowymi parametrami autoryzacji, których używa się kierując docelowe zapytania. Na szczęście w przypadku bardziej popularnych serwisów jakim jest Twitter, istnieje wiele gotowe modułów ułatwiających obsługę zapytań. W szczególności, moduły dedykowane obsłudze konkretnego serwisu (np. python-twitter) automatycznie przeprowadzą złożone operacje „techniczne'' związane z zapytaniem (np. dwustopniową autoryzację).

Przykładowe serwisy API, które mogą być przydatne podczas przetwarzania danych oraz usprawnienia codziennego funkcjonowania przedsiębiorstw:

- GUS — REGON (https://api.stat.gov.pl);
- API Wykazu podatników VAT (https://wl-api.mf.gov.pl);
- NFZ — np. terminy leczenia, lista specjalistów itp. (https://api.nfz.gov.pl);
- Dane meteorologiczne (https://api.meteo.pl);
- API przewoźników czy firm kurierskich (np. https://dhl24.com.pl/webapi2/doc/index.html, https://www.fedex.com/en-ca/resources-tools/api.html, https://www.ups.com/pl/pl/services/technology-integration/online-tools-shipping.page);
- API baz bibliometrycznych (np. Scopus, Web of Science, IEEE Xplore), umożliwiające np. pobieranie danych o publikacjach określonych osób liczbie cytowaniach;
- serwisy pozwalające wysyłać wiadomości SMS oraz serwisy oferujące dostęp do danych z rynków finansowych (w obu wymienionych przypadkach są to zwykle usługi płatne, chociaż oferują bezpłatny okres próbny);

– Ministerstwo Cyfryzacji w „Programie otwierania danych na lata 2021–2027" wskazuje rozwój API jako jeden z priorytetów, m. in. przewiduje się API dla danych z krajowego rejestru sądowego.

Na koniec dodajmy, że wykorzystanie API nie ogranicza się do jednorazowego (lub okazjonalnego) zbierania danych. Wręcz przeciwnie, mając już fragment kodu obsługujący zbieranie danych, można go wykorzystać do regularnej, automatycznej aktualizacji zbioru danych. W przypadku modeli uczenia maszynowego możemy wykorzystać np. dane o bieżącej pogodzie do predykcji (szczegóły takich modeli omawiamy w rozdz. 5). Innym ciekawym rozwiązaniem jest umieszczenie danych z API w serwisie www; w ten sposób możemy np. umieścić automatycznie generowany (a więc także na bieżąco aktualizowany) wykaz publikacji lub liczby cytowań.

Załącznik: wstępna analiza statystyczna ogłoszeń gratka.pl

W rozdz. 3.2 pokazaliśmy w jaki sposób można zebrać dane dotyczące ogłoszeń w serwisie gratka.pl. Poniżej przedstawimy przykład wstępnej obróbki i analizy ogłoszeń sprzedaży samochodu Ford Fiesta zebranych[9] techniką *web scraping*. Pokażemy jak przekształcić zebrane dane do pożądanego formatu (najczęściej liczbowego), a następnie umieścimy dane w ramce danych (wcześniej zadbaliśmy o braki danych, dzięki czemu obserwacje poszczególne listy mają jednakową liczbę elementów i mogą być interpretowane jako kolumny w ramce danych). Na koniec pokażemy kilka wstępnych analiz danych, które można traktować jako sprawdzenie prostych zależności lub weryfikację logicznej poprawności.

Po pierwsze — nie zawsze dane mają właściwy format. Spodziewalibyśmy się, że przebieg będzie wyrażony liczbami. Po drugie — cena zawiera końcówkę 'zł'. Używając operacji na napisach (w tym miejscu musimy nieco wyprzedzić kolejność tematów prezentowanych w książce; Czytelnik znajdzie opis metody .**replace**() w rozdz. 4.1) możemy usunąć tę końcówkę.

```
>>> cena_liczby = [i.replace("zł","") for i in all_cena]
```

Następnie możemy przejść do utworzenia obiektu *pd.DataFrame* (zakładamy, że *pandas* został wcześniej importowany i aliasowany jako 'pd' — zob. rozdz. 2.8), dzięki czemu z łatwością przeprowadzimy dalsze obliczenia czy zapis danych do arkusza CSV. W tym przypadku wiemy, że wszystkie liczby podane w ogłoszeniu powinny być całkowite, więc dobrym zwyczajem jest zadeklarować typ danych w parametrze 'dtype' (w Pythonie, jak wiadomo, taka deklaracja nie jest wymagana, ale w *pandas* niekiedy ułatwia to konwersję ty-

[9] Ściślej rzecz biorąc, pobrano wszystkie ogłoszenia sprzedaży samochodu marki Ford, modelu Fiesta dostępne w dniu 7 marca 2020 r.

pów w przypadkach „niejednoznacznych", np. gdy przed lub po liczbie występują spacje).

```
>>> df1 = pd.DataFrame({'przebieg': dane_przebiegi, 'rok':
dane_rokprodukcji, 'cena': cena_liczby}, dtype=int)
```

Jak pamiętamy z rozdz. 3.2, braki danych co do przebiegu lub roku produkcji oznaczaliśmy liczbami -99. Obserwacje te powinniśmy usunąć ze zbioru danych. Ponieważ będziemy zainteresowani analizą zależności pomiędzy zmiennymi, można to zrobić usuwając wiersze dla obserwacji w których wystąpił choć jeden brak danych. Zastosujemy tutaj indeksowanie *boolean* i metodę nie-równy **.ne()** a rezultat zapiszemy w nowym DataFrame nazwanym 'df_gotowy'. Znak '&' oznacza operację logiczną ORAZ przeprowadzaną na obiektach pandas (dla operacji LUB stosowalibyśmy znak '|' — tzw. pipe).

```
>>> df_gotowy = df1[df1.przebieg.ne(-99) & df1.rok.ne(-99)]
```

Pierwszym etapem analiz będzie policzenie statystyk opisowych.

```
>>> df_gotowy.describe()
            przebieg          rok          cena
count     527.000000   527.000000    527.000000
mean    94011.738140  2012.935484  28373.779886
std     64239.813705     4.508506  16605.814599
min         1.000000  1977.000000   1499.000000
25%     37046.000000  2010.000000  16900.000000
50%     95000.000000  2013.000000  24900.000000
75%    145728.000000  2016.000000  34900.000000
max    275449.000000  2020.000000  75943.000000
```

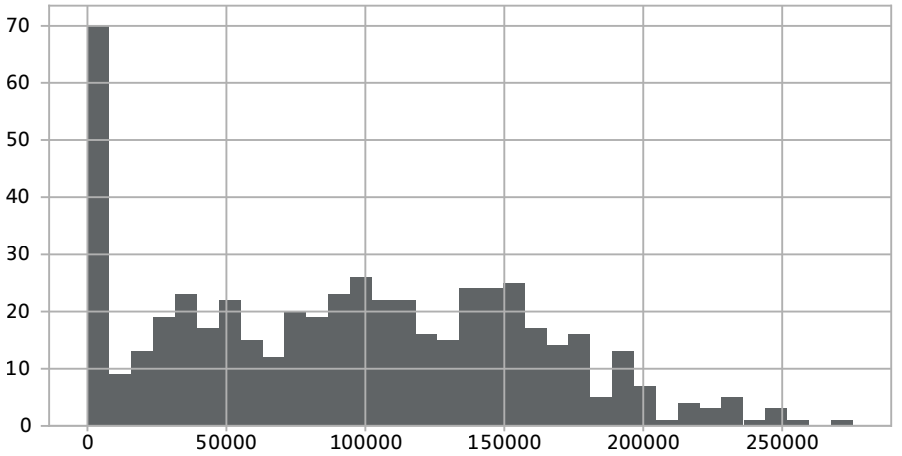
Analiza statystyk opisowych nie pozwala na wyciąganie szczegółowych wniosków (np. zależności pomiędzy różnymi wielkościami). Mimo to zachęcamy Czytelników do przeprowadzania jej w charakterze wstępnej inspekcji danych. Szczególnie zwróćmy uwagę na wartość średnią (`mean`) oraz medianę (`50%`), które pozwalają wyrobić sobie zdanie o skali danego zjawiska. Ułatwi to interpretację, dodatkowo w niektórych przypadkach wywoła refleksję na temat jednostki (np. jeśli średnio przebieg wynosi ok. 94000, to można przyjąć, że jest on wyrażony jest w kilometrach a nie w tysiącach kilometrów). Z kolei wartość minimalna i maksymalna (`min`, `max`) pozwoli nam zweryfikować czy nie ma oczywistych błędów w danych. W szczególności widzimy, że po usunięciu braków danych (tj. obserwacji o przebiegu równym -99) nie występują obserwacje o ujemnym przebiegu, a minimalny przebieg 1 km i maksymalny ok. 275000 km nie budzą wątpliwości. Wreszcie, odchylenie standardowe (`std`) (ewentualnie różnica pomiędzy 75 a 25 percentylem) informuje nas o przeciętnej skali wahań badanej zmiennej.

Oczywiście statystyki opisowe zwrócone przez metodę .**describe**() nie pozwalają uzyskać bardziej precyzyjnej informacji o rozkładzie zmiennej. Więcej

informacji o rozkładzie dostarczy metoda **.hist**(). W ten sposób możemy tworzyć wykres-histogram (a także zapisać ten obraz do pliku graficznego), który wskazuje jak często występują określone przedziały wartości. Przedziały wartości nazywane są „binami". Domyślna liczba binów wynosi 10, co zwykle nie zapewnia precyzyjnego i czytelnego histogramu. Dlatego liczbę binów dobrze jest jednoznacznie wskazać — w poniższym przykładzie dobrą czytelność histogramu uzyskujemy dla 35 binów, ale Czytelnika zachęcamy do poeksperymentowania z parametrem 'bins' we własnych przykładach.

```
>>> histogram_przebieg = df_gotowy.przebieg.hist(bins=35)
>>> histogram_przebieg.figure.show()
```

Rys. 3.3 Histogram przebiegu z ogłoszeń sprzedaży Forda Fiesty



Kolejnym etapem wstępnej analizy danych może być policzenie prostej korelacji liniowej dla wybranych zmiennych. Służy temu metoda **.corr**(), którą możemy zastosować do całego DataFrame bądź do części jego kolumn (tak robimy w poniższym przykładzie). Spójrzmy na przykład:

```
>>> df_gotowy[['przebieg', 'rok']].corr()
          przebieg       rok
przebieg  1.000000 -0.725039
rok      -0.725039  1.000000
```

Otrzymany współczynnik korelacji pomiędzy przebiegiem a rokiem produkcji wynosi ok. -0,73. Wskazuje na silny ujemny związek pomiędzy tymi zmiennymi. Zwracamy Czytelnikowi uwagę, że sam znak korelacji może dostarczyć nam nie tylko informacji o zależności, ale w przypadku gdy kierunek zależności jest „oczywisty" — może służyć jako sprawdzian poprawności danych. W przypadku samochodów ujemna korelacja przebiegu i roku produkcji oznacza, że wraz ze wzrostem roku produkcji (czyli inaczej mówiąc: spadkiem

wieku samochodu) jego przebieg maleje. Trudno byłoby zrozumieć zależność przeciwną, dlatego korelacja ujemna sugeruje, że dane są prawidłowe (a przynajmniej nie dostarcza wątpliwości co do zbioru danych).

4. Podstawy przetwarzania danych tekstowych

Dane tekstowe są reprezentowane przez obiekty typu napis. Przypomnijmy za rozdziałem 2, że typ ten jest niezmienniczym łańcuchem znaków (sekwencją), co oznacza, że raz utworzonego obiektu nie da się modyfikować (ale można utworzyć nowy, zmodyfikowany) oraz że kolejność (pojedynczych znaków) ma znaczenie, dzięki czemu jest to typ iterowalny. Czytelnik poznał już ponadto operacje konkatenacji (czyli łączenia) oraz mnożenia przez liczbę naturalną (czyli zwielokrotnienia) napisów. Wspomniane zostały ponadto tzw. znaki specjalne i sposoby ich zapisu. W tym rozdziale szerzej przyjrzymy się metodom służącym do pracy z tekstem, omówimy tzw. wyrażenia regularne służące do dopasowywania wzorców oraz podstawy przetwarzania języka naturalnego.

4.1 Metody służące do przetwarzania napisów

Jak zostało wspomniane w rozdziale 2, do deklaracji napisów można użyć znaku " (cudzysłowu) oraz ' (apostrofu), przy czym są to zapisy równoważne (przypomnijmy, że znak podwójnej równości jest operatorem porównania):

```
>>> "napis" == 'napis'
True
```

Zauważmy jednocześnie, że zapis z wykorzystaniem wielkiej i małej litery nie jest tożsamy:

```
>>> "napis" == 'Napis'
False
```

Wspomniane sposoby deklaracji napisu są o tyle użyteczne, że pozwalają w łatwy sposób tworzyć napisy zawierające apostrofy (wykorzystując zapis z cudzysłowem) lub cudzysłów (wykorzystując zapis z apostrofami):

```
>>> "Tytuł książki to 'Alicja w Krainie Czarów'. "
"Tytuł książki to 'Alicja w Krainie Czarów'. "
>>> 'Tytuł książki to "Alicja w Krainie Czarów". '
'Tytuł książki to "Alicja w Krainie Czarów". '
```

Gdybyśmy o tym zapomnieli i próbowali wpisać cudzysłów wewnątrz innego cudzysłowu (ograniczającego napis), zostanie wyświetlony komunikat o błędzie:

```
>>> "Tytuł książki to "Alicja w Krainie Czarów". "
SyntaxError: invalid syntax
```

Błędu unikniemy, jeśli przy cudzysłowie (apostrofach), które nie informują o początku i końcu napisu dodamy tzw. znaki ucieczki[1] (ukośniki odwrotne, czyli znaki *backslash*):

```
>>> "Tytuł książki to \"Alicja w Krainie Czarów\". "
'Tytuł książki to "Alicja w Krainie Czarów". '
>>> 'Tytuł książki to \'Alicja w Krainie Czarów\'. '
"Tytuł książki to 'Alicja w Krainie Czarów'. "
```

Co szczególnie ciekawe (i ważne), długości napisów stworzonych z wykorzystaniem znaków ucieczki i bez nich, są takie same:

```
>>> len('Tytuł książki to \'Alicja w Krainie Czarów\'. ')
44
>>> len("Tytuł książki to 'Alicja w Krainie Czarów'. ")
44
```

Innymi słowy, *backslash* występujący wyżej w charakterze znaku ucieczki nie został „policzony". Nie jest bowiem zapisany w pamięci tak jak inne znaki w napisie, a jedynie sygnalizuje „specjalny" sposób traktowania kolejnego znaku. Problem pojawia się, kiedy chcemy użyć „prawdziwego" znaku \, który pojawia się często np. w zapisie ścieżek pliku:

```
>>> "ścieżka pliku to C:\Nowy "
SyntaxError: (unicode error) 'unicodeescape' codec can't de-
code bytes in position 37-38: malformed \N character escape
```

W takiej sytuacji wystarczy zastosować znak ucieczki od „specjalnego" znaczenia ukośnika, czyli w praktyce dwukrotnie użyć ukośnika — po pierwsze, ze względu na fakt, że występuje on w napisie reprezentującym ścieżkę pliku, a po drugie, do tego, aby „wyłączyć" specjalne znaczenie tego znaku — ucieczki.

```
>>> "ścieżka pliku to C:\\Users\\User"
'ścieżka pliku to C:\\Users\\User'
```

[1] Znak ucieczki oznacza „ucieczkę" od domyślnego znaczenia znaku, przez co znak poprzedzony symbolem \ jest traktowany przez interpreter Pythona w „specjalny" sposób.

```
>>> print('ścieżka pliku to C:\\Users\\User')
ścieżka pliku to C:\Users\User
```

Alternatywnie można użyć tzw. surowej (*raw*) notacji, czyli poprzedzić napis literą r (lub R), jak w poniższym przykładzie:

```
>>> r'ścieżka pliku to C:\Users\User'
'ścieżka pliku to C:\\Users\\User'
>>> len(r'ścieżka pliku to C:\Users\User')
30
```

Zauważmy, że napis w notacji surowej został zinterpretowany jako zawierający podwójny ukośnik. Podobnie jak poprzednio, długość napisu uwzględnia pojedynczy, a nie podwójny *backslash*.

Ze względu na fakt, iż w różnych systemach operacyjnych istnieją różne formaty ścieżek, zalecamy następujący sposób ich podawania:

```
>>> import os
>>> path = os.path.join('katalog1','katalog2','katalog3')
```

Wynik taki, jak w powyższym przykładzie, osiągnęlibyśmy poprzez wywołanie:

```
>>> import os
>>> path = os.path.join('C:\\', 'Users', 'User')
>>> napis = "ścieżka pliku to " + path
>>> napis
'ścieżka pliku to C:\\Users\\User'
>>> len(napis)
30
```

W celu usprawnienia działań na napisach dostępne są gotowe metody implementujące najczęściej wykonywane operacje. Poniżej opiszemy działanie podstawowych metod. Jednocześnie zachęcamy Czytelnika do zapoznania się z dokumentacją np. poprzez wywołanie **help**(str) lub **help**(str.nazwa_metody) dla poszczególnych metod, w celu zapoznania się z dodatkowymi ich możliwościami.

Metoda .**find**(fragment) służy do wyszukiwania fragmentów (podnapisów) w zadanym tekście (napisie, na rzecz którego metoda ta jest wywoływana). Zwraca indeks napisu, począwszy od którego zaczyna się szukany fragment (najniższy, czyli pierwszy znaleziony od lewej w przypadku gdy dany fragment zawiera się więcej niż jeden raz w napisie). W przypadku gdy fragment nie jest częścią napisu, metoda ta zwraca wartość -1. W przykładzie poniżej napis to „Python", a wyszukiwany fragment to „on".

```
>>> napis = "Python"
>>> fragment = "on"
>>> napis.find(fragment)
4
```

Zwrócona wartość to 4, ponieważ litera „o" rozpoczynająca szukany fragment znajduje się pod tym indeksem w napisie.

P	y	t	h	o	n
0	1	2	3	4	5
				o	n

```
>>> "Python".find("P")
0
>>> "Python".find("p")
-1
```

Zwróćmy uwagę, że metoda jest wrażliwa na wielkość liter — litera „P" występuje w napisie „Python" na miejscu z indeksem 0, natomiast litera „p" nie występuje w „Python", dlatego zwrócona wartość to -1.

W sposób analogiczny działa metoda .**rfind**(), przy czym zwraca ona nie najniższy, lecz najwyższy indeks, od którego podany fragment zawiera się w napisie. Inaczej mówiąc działa tak samo jak metoda .**find**(), ale przeszukuje napis od końca. W przykładzie poniżej wykorzystano napis „kajak" i literę „k", która rozpoczyna i kończy ten napis. Jak pamiętamy, pierwszy znak napisu ma indeks 0, dlatego metoda .**find**() zwraca 0, natomiast ostatnia litera „k" ma indeks 4, dlatego też tę cyfrę zwraca metoda .**rfind**().

```
>>> "kajak".find("k")
0
>>> "kajak".rfind("k")
4
```

Analogicznie do metod .**find**() i .**rfind**() działają .**index**() i .**rindex**(). Różni je jedynie zwracana wartość w przypadku, gdy szukany fragment nie zawiera się w napisie. W przypadku metod .**index**() i .**rindex**() jest to wyjątek `ValueError: substring not found` (szerzej na temat wyjątków piszemy w rozdz. 3.4), natomiast w przypadku metod .**find**() i .**rfind**() — wartość -1.

```
>>> "kajak".find("0")
-1
>>> "kajak".index("0")
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    "kajak".index("0")
ValueError: substring not found
```

Metoda **.replace**() służy do zastępowania podanych fragmentów (czyli ciągów znaków występujących w napisie) nowym ciągiem znaków. W nawiasie podajemy argumenty w kolejności „stary", „nowy". W przykładzie poniżej została ona wywołana w celu zamiany „na" na napis „po":

```
>>> n = "napis"
>>> n.replace("na", "po")
'popis'
>>> n
'napis'
```

Co ważne, okazało się, że pomimo wykonanej zamiany, zmienna n nadal przechowuje wartość „napis". Przypomnijmy, że jest to typ niezmienniczy, co oznacza, że operacja zwróciła tak naprawdę nowy obiekt, którym można w razie potrzeby nadpisać istniejący:

```
>>> n = n.replace("na", "po")
>>> n
'popis'
```

Metoda **.split**(separator) służy do podziału napisu z użyciem separatora wskazanego jako argument. W przykładzie poniżej zmienna n przechowuje napis składający się z trzech słów oddzielonych spacjami. Jeśli wywołamy metodę **.split**() ze spacją w charakterze separatora, otrzymamy listę słów (z pominięciem spacji):

```
>>> n = "Kot w butach"
>>> n.split(" ")
['Kot', 'w', 'butach']
```

Gdybyśmy jednak w charakterze separatora wskazali inny znak, np. literę „t", otrzymamy inny wynik:

```
>>> n.split("t")
['Ko', ' w bu', 'ach']
```

W poniższej tabeli zaprezentowaliśmy działanie metody **.split**(). Pierwszym elementem zwróconej listy jest „Ko", ponieważ te dwa znaki występują przed pierwszym wystąpieniem separatora (litery „t"). Drugi element to „w bu", czyli fragment napisu pomiędzy pierwszym a drugim wystąpieniem litery „t". Listę kończy „ach", czyli fragment od ostatniego separatora do końca napisu.

K	o	t	w	b	u	t	a	c	h
		↑				↑			

Prześledźmy inny przykład:

```
>>> n = "jajko"
>>> n.split("j")
['', 'a', 'ko']
```

Otrzymaliśmy listę trójelementową, którą rozpoczyna pusty napis. Dzieje się tak dlatego, że pierwszy element listy odpowiada fragmentowi od początku napisu do pierwszego wystąpienia separatora. Nasz napis rozpoczyna się literą „j", którą wskazaliśmy w charakterze separatora, zatem fragment od początku napisu do pierwszej litery „j" to napis pusty. Drugi element listy to litera „a", która w wyjściowym napisie znajduje się pomiędzy pierwszym a drugim separatorem. Ostatni element listy to „ko", czyli fragment od ostatniego separatora do końca napisu.

Metoda .**join**() służy do łączenia (konkatenacji) napisów. Sposób jej wywołania jest następujący: separator.**join**(obiekt_iterowalny). W obiekcie iterowalnym znajdują się elementy, które zostaną połączone z wykorzystaniem separatora. W przykładzie poniżej odwracamy działanie metody .**split**() z ostatniego przykładu. Jako separator podajemy literę „j", natomiast jako obiekt iterowalny — listę zawierającą kolejno: pusty napis, literę „a", napis „ko".

```
>>> 'j'.join(['', 'a', 'ko'])
'jajko'
```

Jak już wspomniano obiektem iterowalnym jest również sam napis. Dlatego też wywołanie metody .**join**() jest możliwe jeśli podamy napis w charakterze argumentu. W przykładzie poniżej użyto spacji jako separatora i pierwszych liter alfabetu. W rezultacie otrzymano napis, w którym podane litery zostały poprzedzielane spacjami.

```
>>> " ". join("abcdef")
'a b c d e f'
```

Metody .**strip**(), .**lstrip**() i .**rstrip**() służą do usuwania białych znaków odpowiednio: z początku i końca, tylko początku i tylko końca napisu. Poniższy przykład dotyczy napisu, który rozpoczyna się tabulatorem, a kończą go dwie spacje oraz znak nowej linii (\n)

```
>>> napis = "Kot w butach  \n"
>>> print(napis) # pusty wiersz wynika z obecności "\n"
w napisie
Kot w butach

>>> napis.strip() # usuwamy białe znaki z początku i końca
napisu
'Kot w butach'
```

Osobna grupa metod służy do formatowania napisów. Są nimi: **.capitalize**(), która zamienia pierwszy znak napisu na wielką literę, **.lower**() (**.upper**()) służąca do zamiany na małe (wielkie) litery wszystkich liter napisu oraz **.title**(), dzięki której każdy wyraz w napisie rozpoczynany jest wielką literą, a kolejne litery są zmienione na małe:

```
>>> "kot".capitalize()
'Kot'
>>> "KoT".lower()
'kot'
>>> "KoT".upper()
'KOT'
>>> "kot w buTach".title()
'Kot W Butach'
```

Metoda **.lower**() jest szczególnie przydatna przy wyszukiwaniu napisów bez względu na wielkość użytych liter. Nawiązując do wcześniejszego przykładu, w którym wyszukiwaliśmy literę „p" w napisie „Python", poniższa składnia pozwala na sprawdzenie czy napis zawiera „p" bez rozróżniania wielkości liter:

```
>>> "Python".lower().find("p")
0
```

Możliwe jest ponadto sprawdzenie w jaki sposób został zapisany napis z wykorzystaniem metod **.istitle**(), **.islower**(), **.isupper**(). Ich działanie nawiązuje do sposobu formatowania napisów omówionych powyżej i sprawdza odpowiednio czy łańcuch znaków jest sformatowany jak tytuł oraz czy jest zapisany małymi/wielkimi literami. Ich działanie ilustrują poniższe przykłady:

```
>>> "Kot".istitle()
True
>>> "KOT".isupper()
True
>>> "kot".islower()
True
```

Metoda **.isalnum**() bada czy wszystkie znaki w napisie są znakami alfanumerycznymi, tzn. czy każdy ze znaków jest liczbą lub literą. Warto zaznaczyć, że znaki białe oraz interpunkcyjne nie są traktowane jako znaki alfanumeryczne, tzn. jeśli napis zawiera co najmniej jedną spację/znak interpunkcyjny, wówczas metoda zwróci wartość *False*. Dla pustego łańcucha zwracana jest również wartość *False*.

```
>>> "".isalnum()
False
>>> "a ".isalnum()
False
```

```
>>> "20019r.".isalnum()
False
>>> "20019r".isalnum()
True
```

Na podobnej zasadzie działają metody .**isalpha**(), która bada czy każdy znak napisu jest literą, .**isdigit**() sprawdzająca czy wszystkie znaki są cyframi oraz .**isspace**() badająca czy każdy znak w napisie jest znakiem białym. Podobnie jak metoda .**isalnum**(), zwracają one wartość *False* w przypadku pustego napisu. Działanie wymienionych metod ilustrują poniższe przykłady:

```
>>> "3".isalpha()
False
>>> "k".isalpha()
True
>>> "3".isdigit()
True
>>> "3".isspace()
False
>>> " ".isalpha()
False
>>> " ".isdigit()
False
>>> " ".isspace()
True
>>> "\n".isspace()
True
>>> "\t".isspace()
True
```

W pracy z napisami użyteczne są ponadto metody .**startswith**(argument) oraz .**endswith**(argument). Sprawdzają one czy napis, na rzecz którego zostały wywołane odpowiednio zaczyna/kończy się napisem wskazanym jako argument metody. Metoda .**count**(fragment) służy natomiast do zliczania wystąpień fragmentu podanego jako argument w napisie, na rzecz którego metoda jest wywoływana.

```
>>> "jajka".endswith("ka")
True
>>> "jajka".startswith("ja")
True
>>> "jajka".count("j")
2
>>> "jajka".count("ja")
1
>>> "jajka".count("jak")
0
```

```
>>> "jajka".count("jajka")
1
>>> "jajka".startswith("jajka")
True
>>> "jajka".endswith("jajka")
True
```

Python dostarcza również narzędzi do formatowania napisów, które mogą być użyteczne zwłaszcza przy pracy z dokumentami o powtarzalnej strukturze. Poniższa funkcja pozwala w łatwy sposób manipulować napisem poprzez zmianę danych:

```
>>> napis = "Pozycja nr {0} to {1}".format("1", "jabłka")
>>> print(napis)
Pozycja nr 1 to jabłka
>>> napis = "Pozycja nr {0} to {1}".format("20", "winogro-
na")
>>> print(napis)
Pozycja nr 20 to winogrona
```

Podobną funkcjonalność możemy uzyskać dzięki klasie **Template**[2]. Wymaga ona importu, a następnie utworzenia obiektu klasy **Template** z argumentem w postaci napisu, w którym będziemy dokonywać zmian. Składnia identyfikatora (czyli fragmentu napisu, który będziemy zmieniać) to ${identyfikator} lub $identyfikator, natomiast do dokonywania zmian służy metoda .**substitute**(), w której argumentach podajemy docelowe wartości (napisy), które mają wystąpić w naszym napisie.

```
>>> from string import Template
>>> t = Template("Pozycja nr ${nr_pozycji} to ${nazwa_pozy-
cji}")
>>> # prostszy zapis z pominięciem nawiasów klamrowych:
>>> t1 = Template("Pozycja nr $nr_pozycji to $nazwa_pozy-
cji")
>>> t.substitute(nr_pozycji = "1", nazwa_pozycji = "jabłka")
'Pozycja nr 1 to jabłka'
>>> t1.substitute(nr_pozycji = "20", nazwa_pozycji = "wino-
grona")
'Pozycja nr 20 to winogrona'
```

Dodatkowym usprawnieniem formatowania napisów jest słownik, który będzie zawierał wszystkie docelowe wartości lub lista zawierająca poszczególne słowniki (korzystamy z *Template* `t` z poprzedniego przykładu):

[2] Istnieją różne metody formatowania napisów, w tym począwszy od wersji 3.6 *f-strings* od *formatted strings*, która pozwala na włączanie wyrażeń Pythona do napisów. Zachęcamy Czytelników do dalszej, samodzielnej lektury w tym zakresie.

```
>>> slownik  = {"nr_pozycji" : "1", "nazwa_pozycji" : "jabł-
ka"}
>>> t.substitute(slownik)
'Pozycja nr 1 to jabłka'
>>> t1.substitute(slownik)
'Pozycja nr 1 to jabłka'
>>> l = [{"nr_pozycji" : "1", "nazwa_pozycji" : "jabłka"},
{ "nr_pozycji" : "20", "nazwa_pozycji" : "winogrona"}]
>>> for i in l:
...     t.substitute(i)
'Pozycja nr 1 to jabłka'
'Pozycja nr 20 to winogrona'
```

Ciekawostka: ze względu na różnorodność możliwości zapisu liczb, w Pythonie są również inne (poza .**isdigit**()) metody ich rozpoznawania:

```
>>> print('\u2154')
⅔
>>> s = '\u2154'
>>> s.isdigit()
False
>>> s.isnumeric()
True
>>> s.isdecimal()
False
```

Metody mają jednak swoje ograniczenia, poniższy przykład przestawia konsekwencje zapisu liczby jako typu napis:

```
>>> "10.1".isdigit()
False
>>> "10.1".isnumeric()
False
>>> "10.1".isdecimal()
False
```

4.2 Wyrażenia regularne

Wyrażenia regularne, inaczej wzorce tekstowe, pozwalają na zdefiniowanie pewnego wzorca za pomocą specjalnych symboli, a następnie sprawdzenie czy dany napis odpowiada podanemu wzorcowi.

W praktycznych zastosowaniach mogą być one użyteczne na przykład do walidacji danych pochodzących z formularza. Przykładowo poprawny adres mailowy musi zawierać ciąg znaków alfanumerycznych i/lub cyfr, symbol @ oraz nazwę domeny (por. np. definicja w standardzie RFC https://tools.ietf.org/html/rfc2822#page-16). Dzięki wyrażeniom regularnym

można zatem sprawdzić czy dane wpisane przez użytkownika odpowiadają typowemu adresowi mailowemu.

Jak już wspominano, wyrażenia regularne wykorzystują symbole specjalne, tzw. metaznaki, czyli takie, którym przypisano dodatkowe znaczenie. Przykładowo znak „.", czyli kropka, w kontekście wyrażeń regularnych reprezentuje dowolny znak (z wyjątkiem znaku nowej linii). Aby wykorzystać moduł o nazwie *re* odpowiadający za dopasowanie wzorców należy go zaimportować, ponieważ nie jest on domyślnie ładowany do pamięci. Następnie użyjemy metody *re*.**findall**(wzorzec, napis), która zwróci listę wszystkich dopasowań zadanego wzorca (podanego jako pierwszy argument) do napisu wskazanego w drugim argumencie. W przykładzie poniżej naszym wzorcem jest po prostu kropka, a napis, który przeszukujemy pod kątem dopasowania do wzorca to napis „00-005 Łódź".

```
>>> import re
>>> re.findall("." , "00-005 Łódź")
['0', '0', '-', '0', '0', '5', ' ', 'Ł', 'ó', 'd', 'ź']
```

Do podanego wzorca pasuje każdy znak, a zatem dopasowania, które są elementami zwróconej listy to pojedyncze cyfry, myślnik, spacja oraz pojedyncze litery. Jeśli natomiast przeszukiwany napis będzie pusty, zwrócona lista dopasowań będzie pusta.

```
>>> re.findall("." , "")
[]
```

W kolejnym przykładzie szukany wzorzec to litera „o" poprzedzona dowolnym znakiem. Zwracamy uwagę, że lista dopasowań zawiera jedynie fragmenty napisu pasujące do wzorca, a nie całe słowa zawierające wzorzec:

```
>>> re.findall(".o", "programowanie w języku Python")
['ro', 'mo', 'ho']
```

Istnieje również możliwość wskazywania zakresu znaków za pomocą nawiasów kwadratowych. Zapis [pyt] wskazuje na dowolny znak z podanych w nawiasie, z kolei zapis [^pyt] na dowolny znak z wyjątkiem tych, podanych w nawiasie (zwracamy uwagę na inne znaczenie symbolu „^" w przypadku wykorzystania go w nawiasie kwadratowym).

```
>>> re.findall("[pyt]", "Python")
['y', 't']
>>> re.findall("[^pyt]", "Python")
['P', 'h', 'o', 'n']
```

Możliwy jest ponadto zapis z wykorzystaniem myślnika, wskazujący na zakres, np. dowolna cyfra to [0–9], a dowolna mała litera to [a–z]. Dla polskich znaków jest to zakres [a–ż]:

```
>>> re.findall("[a-z]", "język Python")
['j', 'z', 'y', 'k', 'y', 't', 'h', 'o', 'n']
>>> re.findall("[a-ż]", "język Python")
['j', 'ę', 'z', 'y', 'k', 'y', 't', 'h', 'o', 'n']
```

Zazwyczaj występuje wrażliwość na wielkość liter, czyli zakres [a–z] nie pokrywa się z [A–Z].

Wskazanie zakresu możliwe jest również dla cyfr, np.:

```
>>> re.findall("[0-9]", "Python 3.8")
['3', '8']
```

Inne wbudowane klasy znaków przedstawiono w Tab. 4.1.

Tab. 4.1 Podstawowe klasy znaków w wyrażeniach regularnych

Symbol	Wyjaśnienie
\w	dowolny znak alfanumeryczny, co odpowiada [a-zA-Z0-9_]
\W	dowolny znak niealfanumeryczny, czyli [^a-zA-Z0-9_]
\d	cyfra (*decimal digit*) symbol równoważny [0-9]
\D	dowolna niecyfra, czyli to samo co [^0-9]
\s	nowa linia, spacja, tabulacja — tzw. białe znaki, czyli odpowiednik [\t\n\r\f\v]
\S	znak niebiały

Porównajmy wywołania:

```
>>> re.findall("\d", "Python 3.8")
['3', '8']
>>> re.findall("\d.", "Python 3.8")
['3.']
```

W pierwszym przypadku jako wzorzec wskazaliśmy „\d", czy dowolną cyfrę, stąd też dopasowania to 3 oraz 8, czyli lista składająca się z 2 elementów.

Z kolei wzorzec „\d." oznacza dowolny znak („.") poprzedzony cyfrą („\d"), dlatego też została zwrócona lista z jednym elementem („3.").

Ciekawostka: jak wyszukać podwójny *backslash* pojawiający się na przykład w ścieżkach plików?

Jak już wspominano, aby „uciec" od specjalnego znaczenia *backslasha*, Python wymaga użycia „\\", ale żeby móc użyć tegoż wyrażenia jako literału Pythona, należy dodatkowo „uciec" od specjalnego znaczenia każdego z *backslashy* osobno, czyli de facto, aby dopasować jeden *backslash*, należy użyć wyrażenia „\\\\".

```
>>> print(re.findall("\\\\", "C:\\\\"))
['\\', '\\'] # dopasowanie pojedynczego backslasha
```

```
>>> l = re.findall("\\\\", "C:\\\\")
>>> len(l[0]) # sprawdzamy długość dopasowania
1
>>> print(re.findall("\\\\\\\\", "C:\\\\"))
['\\\\']
>>> l = re.findall("\\\\\\\\", "C:\\\\")
>>> len(l[0])
2
```

Inne znaki specjalne oraz ich znaczenie objaśniono poniżej:

„^" (daszek) oznacza dopasowanie wzorca jedynie jeśli rozpoczyna on tekst. Przykładowo wzorzec składający się z litery „P" i dowolnego znaku zostanie dwukrotnie dopasowany w napisie „Programowanie w języku Python":

```
>>> re.findall("P.", "Programowanie w języku Python")
['Pr', 'Py']
```

Jeśli jednak poprzedzimy go daszkiem (^) dopasowany zostanie tylko pierwszy — rozpoczynający napis

```
>>> re.findall("^P.", "Programowanie w języku Python")
['Pr']
```

„$" (znak dolara) oznacza dopasowanie na końcu napisu. Rozważmy wzorzec składający się z litery „n" poprzedzonej dowolnym znakiem, czyli „.n". Pojawia się on dwukrotnie w napisie „Programowanie w języku Python", jednak kiedy dodamy znak dolara, czyli wzorcem będzie „.n$", zwrócone zostanie jedynie dopasowanie kończące napis.

```
>>> re.findall(".n", "Programowanie w języku Python")
['an', 'on']
>>> re.findall(".n$", "Programowanie w języku Python")
['on']
```

Istnieje również możliwość wskazania ilokrotnie ma się pojawić określony wzorzec. Służą do tego wyrażenia nazywane kwantyfikatorami. Są to wartości wspinane w nawiasach klamrowych, „*" (gwiazdka) , „+" (plus) oraz „?" (znak zapytania):

{m,n} — określa, że wyrażenie ma być powtórzone od m do n krotnie, na przykład wyrażenie „b{2,3}" wskazuje na dopasowanie wzorca (w przykładzie litery „b") powtórzonego od 2 do 3 razy, czyli ciągi „bb" oraz „bbb".

```
>>> re.findall("b{2,3}", "ababbabbabbb")
['bb', 'bb', 'bbb']
```

W zapisie tym możliwe jest podanie tylko jednego ograniczenia, tzn. {m,} lub {,n} wskazującego odpowiednio dolną i górną liczbę powtórzeń, przy czym określając górną granicę dopuszczamy puste dopasowania:

```
>>> re.findall("b{,2}", "ababbabbabbb")
['', 'b', '', 'bb', '', 'bb', '', 'bb', 'b', '']
```

Dlaczego w wyniku dopasowania otrzymaliśmy puste znaki? Otóż wskazaliśmy, że szukamy litery „b" powtórzonej do 2 razy, co w szczególności oznacza pojedynczą literę „b", ale również jej brak. Innymi słowy, pusty napis również pasuje do tak zadanego wzorca:

```
>>> re.findall("b{,2}", "")
['']
```

Tak zwane puste dopasowania (*empty matches*) są zwracane w wyniku wywołania, pod warunkiem, że nie graniczą bezpośrednio z kolejnym dopasowaniem. Skąd wynika liczba pustych dopasowań, które otrzymaliśmy? Prześledźmy poniższy przykład. Pierwsze (puste) dopasowanie to pusty znak między początkiem napisu oraz literą „a", drugie puste dopasowanie to pusty znak po literze „b", ale przed końcem napisu:

```
>>> re.findall("b{,2}", "ab")
['', 'b', '']
```

Jak już wspomnieliśmy puste dopasowania są zwracane jeśli nie występują bezpośrednio przed kolejnym dopasowaniem, dlatego też zmiana napisu z „ab" na „b" sprawi, że zniknie pierwsze puste dopasowanie:

```
>>> re.findall("b{,2}", "b")
['b', '']
```

W przykładzie poniżej wskazujemy dolne ograniczenie powtórzeń zadanego wzorca:

```
>>> re.findall("b{2,}", "ababbabbabbb")
['bb', 'bb', 'bbb']
```

Możliwe jest również wskazanie dokładnej liczby powtórzeń w nawiasie klamrowym, na przykład:

```
>>> re.findall("b{2}", "ababbabbabbb")
['bb', 'bb', 'bb']
```

Znaczenie pozostałych kwantyfikatorów jest następujące:

– „*" (gwiazdka) określa powtórzenie danego wzorca 0 lub więcej razy (co jest tożsame z zapisem {0,});
– „+" (plus) określa powtórzenie danego wzorca 1 lub więcej razy (co najmniej 1 raz) (tożsame z zapisem{1,});
– „?" (znak zapytania) oznacza powtórzenie 0 lub 1 raz (tożsame z zapisem {0,1}).

Prześledźmy ich działanie na przykładach:

```
>>> re.findall("b*", "ababbabbabbb")
['', 'b', '', 'bb', '', 'bb', '', 'bbb', '']
>>> re.findall("b+", "ababbabbabbb")
['b', 'bb', 'bb', 'bbb']
>>> re.findall("b?", "ababbabbabbb")
['', 'b', '', 'b', 'b', '', 'b', 'b', '', 'b', 'b', 'b', '']
```

Zwracamy uwagę, że kwantyfikatory „*", „+" oraz „?" są zachłanne (*greedy*), co oznacza, że starają się dopasować możliwie najdłuższy fragment. Zauważmy też, że znak kwantyfikatora odnosi się do tylko jednego znaku — tego, który stoi bezpośrednio przed nim:

```
>>> re.findall("ab*", "abababbabb")
['ab', 'ab', 'ab', 'abb']
>>> re.findall("ab{2,3}", "abababbabb")
['abb']
```

Aby wskazać cały powtarzany fragment, należy wpisać fragment w nawias zwykły i poprzedzić symbolem „?:", czyli użyć składni (?:szukany_fragment).

```
>>> re.findall("(?:ab){2,3}", "abababbabb")
['ababab']
>>> re.findall("(?:ab)*", "abababbabb")
['abababab', '', '']
```

Uwaga: jeśli pominiemy „?:", ale pozostawimy nawiasy, to przy wywołaniu metody **.findall**() uzyskamy jedynie ostatnio dopasowany fragment, czyli pojedyncze „ab". Jednak wzorzec będzie nadal odpowiadał odpowiedniemu napisowi, co możemy sprawdzić w tym przypadku wywołując metodę **.search**().

```
>>> re.findall("(ab){2,3}", "abababbabb")
['ab']
>>> re.search("(ab){2,3}", "abababbabb")
<_sre.SRE_Match object; span=(0, 6), match='ababab'>
```

Żeby jeszcze lepiej przyjrzeć się tej zasadzie prześledźmy:

```
>>> re.findall("(?:\d\d)*", "12 34 56 7 8900")
['12', '', '34', '', '56', '', '', '', '8900', '']
>>> re.findall("(\d\d)*", "12 34 56 7 8900")
['12', '', '34', '', '56', '', '', '', '00', '']
```

Zwracamy ponadto uwagę, że metoda **.search**() przeszukuje cały napis w poszukiwaniu podanego wzorca, jednak zwraca jedynie pierwsze jego wystąpienie:

```
>>> re.search("(\d\d)*", "12 34 56 7 8900")
<_sre.SRE_Match object; span=(0, 2), match='12'>
```

Wróćmy do opisu pozostałych oznaczeń:

– „\b" to granica słowa (początek lub koniec) — np. „.+n\b" oznacza dowolne słowo (składające się z dowolnego znaku występującego 1 lub więcej razy) kończące się na literę „n";

```
>>> re.findall(".+n\\b", "Python")
['Python']
```

– „|" — lub — operator pozwalający łączyć wyrażenia regularne. W najprostszym przypadku wyrażeniem regularnym jest pojedyncza litera. Oznacza to, że wzorzec „o|n" będzie pasował do „o" lub do „n", przy czym nie jest on zachłanny — zwróci pierwsze możliwe dopasowanie wyrażenia regularnego, przy czym w pierwszej kolejności będzie wyszukiwał wzorca znajdującego się najbardziej na lewo w ciągu wykorzystującym „|". W przykładzie „o|n", najpierw będzie szukał litery „o", a dopiero w razie gdy jej nie znajdzie, litery „n".

Porównajmy wywołania:

```
>>> print(re.findall("o|n", "Python"))
['o', 'n']
```

Pierwszym dopasowaniem była litera 'o', a kolejnym 'n'.

```
>>> print(re.findall("o*|n", "Python"))
['', '', '', '', 'o', '', '']
```

Tym razem nasz wzorzec to litera „o" występująca dowolną liczbę razy (w tym zero) lub litera „n" Dlatego też wyrażenie „o*" jest za każdym razem dopasowane i z tego powodu „n" nie jest w ogóle brane pod uwagę.

Uwaga: znaki specjalne nie działają „zawsze", tzn. np. jeśli „$" umieścimy w nawiasie kwadratowym: [as$], wówczas wyrażenie to dopasuje się do dowolnego znaku z podanych w nawiasie, czyli również do „$", natomiast co do zasady „$" zwykle ma tzw. specjalny charakter (wyrażający koniec napisu).

Praktyczne przykłady wyrażeń regularnych:

– kod pocztowy w Polsce składa się z 2 cyfr kreski i kolejnych 3 cyfr, co odpowiada wyrażeniu [0-9]{2}-[0-9]{3} lub inaczej \d{2}-\d{3};
– numer PESEL to ciąg jedenastu cyfr: [0-9]{11} co można zapisać najprościej jako \d{11}. Dodatkowych warunków dostarcza wiedza o poszczególnych cyfrach[3]: 6 pierwszych pozycji to data urodzenia w postaci rrmmdd, przy czym dla osób urodzonych w XXI w. do liczby miesiąca dodaje się 20, cyfry na pozycjach od 7 do 10 oznaczają numer serii, a ostatnia liczba to tzw. cyfra

[3] Przykład odnosi się do urodzonych w XX i XXI wieku.

kontrolna. Stąd też nieco dokładniejszym „przybliżeniem" numeru PESEL będzie wyrażenie [0-9]{2}[0-3]{1}[0-9]{1}[0-3]{1}[0-9]{6}. We wzorcu tym dwa pierwsze znaki to dowolna liczba roku (co odpowiada zakresowi 00 do 99), następnie dla pierwszej cyfry miesiąca warunek, że może być ona równa 0, 1, 2 lub 3 (2 i 3 dla osób urodzonych w XXI wieku), dowolna druga cyfra miesiąca, 0, 1, 2 lub 3 jako pierwsza cyfra dnia urodzenia, dowolne cyfry na dalszych pozycjach;

– numer telefonu komórkowego zapisany jako ciąg 9 cyfr lub ciągi 3 cyfr przedzielane myślnikami ([0-9]{3}-?[0-9]{3}-?[0-9]{3});
– oznaczenie numeru faktury przy założeniu, że składa on się z napisu „faktura", „fa", „fv", a dalej jej numeru poprzedzonego napisem „numer" lub „nr" i ewentualnie znakiem „:". Sam numer faktury może mieć różne formaty, dlatego przyjmiemy jedynie, że składa się z co najmniej 4 cyfr. Dodatkowo zakładamy, że pomiędzy poszczególnymi elementami mogą (lub nie) wystąpić białe znaki. Dopuszczamy zapis zarówno wielkimi, jak i małymi literami, dlatego też wykorzystujemy flagę re.IGNORECASE:

```
>>>re.search(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*\d{4,}',
"Faktura nr 1234", re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 15), match='Faktura nr
1234'>
```

Możliwe jest ponadto wyłuskanie tylko części z dopasowanego wzorca. Umożliwiają to wyrażenia zapisane w nawiasach zwykłych, do których po dopasowaniu można się odwołać poprzez tzw. odwołanie wsteczne (*backreference*) w postaci „\nr_grupy". Prześledźmy poniższy przykład. Korzystamy z metody .**sub**(), która przyjmuje 3 argumenty, z których pierwszym jest wzorzec z poprzedniego przykładu, przy czym tym razem \d{4,} zapisaliśmy w nawiasie, drugim jest wyrażenie zastępujące — u nas odwołanie wsteczne, a trzecim — napis, na którym zastąpienia są wykonywane. W kolejnych krokach odwołujemy się do kolejnych dopasowanych grup:

```
>>> re.sub(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*(\d{4,})',
r'\1', "faktura nr 1234")
'faktura'
>>> re.sub(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*(\d{4,})',
r'\2', "faktura nr 1234")
'ktura'
>>> re.sub(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*(\d{4,})',
r'\3', "faktura nr 1234")
'nr'
>>> re.sub(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*(\d{4,})',
r'\4', "faktura nr 1234")
'1234'
```

Jak widzimy, pierwszą grupę stanowi napis faktura, co odpowiada zewnętrznemu nawiasowi w „(fa(ktura)?|fv)", drugim — dopasowanie z nawiasu wewnętrznego, czyli „ktura". Trzecia grupa to napis „nr", co odpowiada „(nr|numer)?", a czwarty to numer faktury, który określiliśmy jako ciąg co najmniej 4 cyfr, czyli „(\d{4,})".

Zauważmy ponadto, że dopasowanie grupy może być napisem pustym. Przykładowo odwołajmy się do grupy trzeciej (czyli „(nr|numer)?") i jednocześnie podajmy napis „faktura 1234".

```
>>> re.sub(r'(fa(ktura)?|fv)\s+(nr|numer)?\s*:?\s*(\d{4,})',
r'\3', "faktura 1234")
''
```

– sygnatura sprawy karnej składająca się z cyfry rzymskiej, pojedynczego białego znaku, litery „K", ponownie — białego znaku, ciągu cyfr, symbolu „/" oraz dwóch cyfr: [I,V,X]{1,5}\s[K]\s\d+/\d{2}.

4.3 Wprowadzenie do przetwarzania języka naturalnego

W tej części zaprezentowane zostaną narzędzia służące do analizy danych tekstowych. Dzięki nim możliwa będzie ekstrakcja i przetwarzanie informacji zawartych np. w mediach społecznościowych, na blogach czy w e-mailach. Przedstawione zostaną metody służące do „przełożenia" danych tekstowych na format liczbowy tak, aby mogły być przedmiotem analizy prowadzonej z wykorzystaniem metod uczenia maszynowego.

Automatyczne przetwarzanie danych w postaci tekstowej jest zadaniem szczególnie trudnym m.in. ze względu na ogromny (i wciąż przyrastający) wolumen, a także na brak ich ustrukturyzowania. Z powodu złożoności omawianego zagadnienia, skupimy się na podstawowych metodach pracy z tekstem. Omówimy m.in. odczyt/zapis danych tekstowych z/do pliku, podział tekstu, metody sprowadzania słów do postaci podstawowej oraz reprezentację tekstu. Jednocześnie odsyłamy Czytelnika do lektury pozycji w całości poświęconych przetwarzaniu języka naturalnego w Pythonie: Sarkar (2016), Bird, Klein, Loper (2009), rozdziału 6 w książce Hearty (2016), a także bardziej ogólnych, dotyczących przetwarzania tekstu: Jurafsky, Martin (2009), Manning, Raghavan, Schütze (2009) oraz Mykowiecka (2007). Do zwięzłego omówienia zastosowań analizy tekstu oraz głównych wątków badawczych odsyłamy Czytelnika do Gładysz (2016).

W niniejszej książce uwagę ograniczono do narzędzi przetwarzania danych tekstowych w języku angielskim. Zachęcamy jednocześnie Czytelnika do zapoznania się z dostępnymi zasobami służącymi przetwarzaniu tekstu w języku polskim, dostępnymi na stronach: http://www.nlp.pwr.wroc.pl, http://zil.ipipan.waw.pl.

4.3.1 Odczyt danych z pliku

Żeby móc przetwarzać dane tekstowe, musimy je najpierw wczytać. Załóżmy na chwilę, że interesujące nas dane są zapisane w pliku tekstowym. Do otwierania plików służy polecenie **open**, w którym jako pierwszy argument wskazujemy nazwę pliku, a jako drugi — tryb, którego wartością domyślną jest „r'' (*read*, czyli odczyt pliku). Inne, możliwe wartości tego argumentu to m.in. „w'' (*write*) oznaczające otwarcie w celu zapisu oraz „a'' (*append*) służące do dopisywania tekstu na końcu pliku. Argument r+ wskazuje natomiast jednoczesne zapisywanie i odczyt z pliku. W przypadku próby zapisu do pliku, który nie istnieje, automatycznie tworzony jest plik ze wskazaną nazwą. Uwaga: otwarcie pliku w trybie „w'' (lub „w+'') oznacza utratę wcześniejszej jego zawartości.

Dobrą praktyką jest używanie w celu otwarcia pliku instrukcji **with open** … **as** …, która zapewni zamknięcie pliku (niezależnie od ewentualnego wyjątku przy jego przetwarzaniu). Innymi słowy, nie będziemy musieli pamiętać o konieczności zamknięcia pliku po wykonaniu jego przetwarzania[4].

W kodzie poniżej otwieramy plik do zapisu (tworzymy nowy jeśli plik o podanej nazwie nie istnieje) i zapisujemy w nim napis „Ala ma kota''.

```
>>> with open('output.txt', 'w') as f:
...     f.write('Ala ma kota')
```

Funkcja **open**() zwraca obiekt pliku (powyżej nazwany literą f). Pozwala on na dostęp do metod oraz atrybutów pliku. Składania polecenia służącego do zapisu do pliku to plik.**write**("napis"). Innymi słowy — dla obiektu plik wywołujemy metodę .**write**(), której argumentem jest napis, który chcemy zapisać do pliku. Załóżmy następnie, że chcemy do pliku dopisać kolejną linię, a następnie wczytać pierwszą. W tym celu otwieramy plik w trybie „a+'', następnie zapisujemy znak nowej linii oraz treść kolejnej linii. Aby wczytać pierwszą linię, musimy wrócić na początek pliku wywołując **seek**(0). Do odczytu linii tekstu w pliku służy metoda **readline**(). Z kolei .**read**() wczytuje plik od bieżącego położenia do końca. Prześledźmy te operacje na poniższych przykładach:

```
>>> with open('output.txt', 'a+') as f:
...     f.write('\n')  #zapisujemy znak nowej linii
...     f.write('a kot ma Alę \n')  # oraz kolejną linię
tekstu
...     f.seek(0) # wracamy na początek pliku
...     print(f.readline()) # wczytujemy pierwszą linię pliku
```

Otrzymany napis to „Ala ma kota". Dla porównania prześledźmy te same kroki zakończone wywołaniem f.**read**().

[4] Gdybyśmy po prostu otworzyli plik np. f = **open**('output.txt', 'r'), wówczas po wykonaniu przetwarzania musielibyśmy go zamknąć wywołując metodę f.**close**().

```
>>> with open('output.txt', 'a+') as f:
...     f.write('\n')
...     f.write('a kot ma Alę \n')
...     f.seek(0)
...     print(f.read())
```

Zgodnie z oczekiwaniami, otrzymaliśmy informację o całej bieżącej zawartości pliku. W przykładzie powyżej po prostu wypisaliśmy ją na ekranie. Możliwe jest również zaczytanie zawartości pliku do zmiennej typu napis (w przykładzie poniżej nazwaliśmy tę zmienną `dane`), a następnie odwołanie się do części znaków:

```
>>> with open('output.txt', 'r') as f:
...     dane = f.read()
>>> print ("Pierwsze 5 znaków pliku:", dane[0:5])
```

Zwróćmy uwagę, że metody .**read**() oraz .**readline**() zwracają obiekty typu napis. Do opisu metod związanych z listami: .**readlines**() oraz .**writelines**() odsyłamy czytelnika do dokumentacji biblioteki dostarczanej razem z Pythonem.

Ciekawostka: obiekt pliku jest jednocześnie iteratorem, co daje możliwość użycia pętli przechodzącej po kolejnych liniach pliku:

```
>>> for line in f:
...     print(line)
```

4.3.2 Podstawy pracy z tekstem

W poniższych przykładach będziemy wykorzystywali popularną bibliotekę zawierającą narzędzia służące do pracy z tekstem — *nltk* (*Natural Language Toolkit*). Pakiet ten wymaga pobrania i instalacji, do szczegółów odsyłamy Czytelnika na stronę https://www.nltk.org/install.html. Jednocześnie polecamy podręcznik omawiający możliwości pakietu *nltk* dostępny pod adresem https://www.nltk.org/book.

Jak już wspomnieliśmy, podstawowym wyzwaniem w pracy z danymi tekstowymi jest ich umiejętne przełożenie na „język liczb". Sposób reprezentacji tekstu w dużej mierze zależy od celu analizy, a także od specyfiki danych, jednak można mówić o pewnych typowych zadaniach składających się na przetwarzanie danych tekstowych. Poniżej dokonujemy syntetycznego opisu poszczególnych zadań, które często składają się kolejne etapy pracy.

1. Tokenizacja — polega na podziale tekstu na mniejsze części — najczęściej chodzi o poszczególne słowa, liczby czy znaki interpunkcyjne.

```
>>> import nltk
>>> dane = "I'm Sorry, I'll Read that Again"
>>> nltk.word_tokenize(dane)
```

```
['I', "'m", 'Sorry', ',', 'I', "'ll", 'Read', 'that', 'Aga-
in']
```

2. Czyszczenie tekstu — przykładowe czynności mogą obejmować m.in. usuwanie białych znaków oraz słów nieistotnych, usuwanie znaków interpunkcyjnych czy ujednolicenie wielkości liter. W zależności od potrzeb mogą to być również takie zadania jak zmiana formatu daty czy rozwijanie bądź usuwanie skrótów. Zakres i sposób czyszczenia tekstu zależy w dużej mierze od charakteru danych, wyjściowej ich postaci oraz celu przetwarzania.
3. *Stemming* — polega na sprowadzeniu wyrazów do ich „*stemów*", czyli rdzeni. W rezultacie słowa takie, jak na przykład: *run*, *runner*, *running*, *runs*, *runny* zostaną sprowadzone do rdzenia, którym jest *run*. Prześledźmy działanie dwóch *stemmerów* dostępnych w pakiecie *nltk* dla tej samej listy słów.

```
>>> lista_slow = ['run', 'runner', 'running', 'runs',
'runny']
>>> from nltk.stem import PorterStemmer
>>> Porter_stemmer = PorterStemmer()
>>> print([Porter_stemmer.stem(w) for w in lista_slow])
['run', 'runner', 'run', 'run', 'runni']
>>> from nltk.stem import LancasterStemmer
>>> Lancaster_stemmer = LancasterStemmer()
>>> print([Lancaster_stemmer.stem(w) for w in lista_slow])
['run', 'run', 'run', 'run', 'runny']
```

Zwróćmy uwagę na składnię. Użycie *stemmerów* wymagało ich importu. Następnym krokiem było utworzenie obiektu *stemmera*, dzięki któremu możliwe było wywołanie metody .**stem**() dla poszczególnych słów z naszej listy.

Na powyższym przykładzie widzimy, że działanie *stemmerów* dało różne rezultaty, przy czym żaden z nich nie jest dokładnie tym, czego oczekiwaliśmy. Dlaczego tak się stało? Otóż działanie *stemmerów* jest oparte na zbiorze reguł (*rule-based*) określających sposoby zamiany końcówek, na przykład w *stemmerze* Portera końcówka „ies" zamieniana jest na końcówkę „i", natomiast końcówka „s", sugerująca liczbę mnogą, jest usuwana. Prześledźmy te reguły na poniższych przykładach:

```
>>> print(Porter_stemmer.stem("bodies"))
bodi
>>> print(Porter_stemmer.stem("ears"))
ear
```

Do szerszych informacji odsyłamy do opisu algorytmu Portera (1980) oraz strony poświęconej algorytmowi http://www.tartarus.org/~martin/PorterStemmer. Zachęcamy również Czytelników do zapoznania się z dokumentacją *stemmerów* (dla różnych języków) dostępnych w *nltk* https://www.nltk.org/api/nltk.stem.html.

Nieco odmiennym pojęciem od *stemmingu* jest *lematyzacja* (hasłowanie). Polega ona na dopasowaniu do podanego słowa odpowiadającego mu zapisu słownikowego. Z jednej strony jest więc podobna do *stemmingu* w tym sensie, że sprowadza wyraz do formy podstawowej, z drugiej — jest zadaniem bardziej złożonym, wymagającym użycia słownika. W przykładzie poniżej posłużymy się *WordNetLemmatizer* wykorzystującym *WordNet*. Do dalszych informacji dotyczących tego zasobu odsyłamy czytelników np. do Maziarz, Piasecki, Rudnicka (2014) oraz Miller, Beckwith, Fellbaum, Gross, Miller (1993/1990), Miller (1993/1990), Gross, Miller (1993/1990), Fellbaum (1990), Beckwith, Miller, Tengi (1993).

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> tekst = ["Ala", "has", "cats"]
>>> for t in tekst:
...     print(lemmatizer.lemmatize(t))
Ala
ha
cat
```

Podobnie jak *stemmery*, narzędzie to wymaga importu, a następnie zainicjowania. Na rzecz utworzonego obiektu (*lemmatizer*) wywołujemy metodę .**lemmatize**() z argumentem w postaci listy słów.

Jak widzimy narzędzie to nie zadziałało idealnie. Stało się tak, ponieważ zgodnie z opcją domyślną (`pos = 'n'`) każde ze słów jest traktowane jak rzeczownik (i dla rzeczowników wyniki były poprawne). Uzyskany rezultat zmieni się jeśli w charakterze drugiego argumentu metody .**lemmatize**() podamy `pos = 'v'` dla czasownika have)

```
>>> print(lemmatizer.lemmatize("has", pos = 'v'))
have
```

Możliwe jest także podawanie parametru informującego o części mowy (*part of speech*) za pomocą narzędzi *nltk* (*pos_tag*), ale jego omówienie, podobnie jak możliwości *WordNet*, wykracza poza zakres książki. Dla opisu innych użytecznych funkcjonalności pakietu *nltk* odsyłamy Czytelnika do https://www.nltk.org/howto.

4. Reprezentacja tekstu — aby tekst mógł być analizowany z wykorzystaniem modeli uczenia maszynowego, musimy dokonać jego wektoryzacji, czyli konwersji na język liczb. Omówimy poniżej podstawowe sposoby reprezentacji tekstu.

Załóżmy przez chwilę, że nasz korpus (czyli zbiór dokumentów) składa się jedynie z 3 tekstów, przy czym pojedynczy dokument będzie jednozdaniowy. Na potrzeby przykładu przyjmijmy również, że świadomie będziemy reprezentować tekst bez jego wstępnego przetworzenia (tzn. nie będziemy usuwać

stopwords[5], ani sprowadzać słów do form podstawowych). Nasze dokumenty to:

```
doc1 = "two alligators, two alpacas and three alligators"
doc2 = "two alligators, two alpacas and two ants"
doc3 = "five ants"
```

Łączymy następnie je w korpus:

```
>>> docs = [doc1, doc2, doc3]
```

Następnie importujemy i inicjujemy *CountVectorizer* i dzięki stworzonemu obiektowi wywołujemy metodę .**fit_transform**() z jednym argumentem — korpusem tekstów. Zwraca ona obiekt (macierz) będący numeryczną reprezentacją naszego korpusu.

```
>>> from sklearn.feature_extraction.text import CountVecto-
rizer
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(docs)
```

Strukturę macierzy X poznamy wypisując najpierw nazwy jej kolumn, a następnie zawartość:

```
>>> print(vectorizer.get_feature_names())
['alligators', 'alpacas', 'and', 'ants', 'five', 'three',
'two']
>>> print(X.toarray())
[[2 1 1 0 0 1 2]
 [1 1 1 1 0 0 3]
 [0 0 0 1 1 0 0]]
```

Zauważmy, że nazwy kolumn to poszczególne słowa z dokumentów w naszym korpusie (uporządkowane alfabetycznie). Macierz zawiera z kolei liczbę wystąpień danego słowa w danym dokumencie. Taka reprezentacja tekstu nosi nazwę reprezentacji ilościowej (Cieślewicz, Pelikant, 2009). Poniżej przedstawiamy macierz w bardziej czytelnej formie.

	„alligators"	„alpacas"	„and"	„ants"	„five"	„three"	"two"
doc1	2	1	1	0	0	1	2
doc2	1	1	1	1	0	0	3
doc3	0	0	0	1	1	0	0

[5] Terminem *stopwords* określane są wyrazy, które występują w danym języku szczególnie często i dlatego też nie mają większego znaczenia w analizie. Przykładami takich słów w języku polskim mogą być: *„że"*, *„i"*, *„oraz"*. Szerzej piszemy na ten temat dalej w niniejszym rozdziale.

Przykładowo w doc2 (środkowy wiersz w tabeli) występują słowa (w kolejności alfabetycznej): *alligators*, *alpacas*, *and*, *ants* (jednokrotnie) oraz trzykrotnie słowo *two*. Pozostałe słowa nie występują wcale, o czym informują zerowe wartości.

Przedstawione wyżej podejście nosi nazwę worka (multizbioru) słów (*bag of words*) lub unigramu, ponieważ przedmiotem analizy są pojedyncze słowa i ich wystąpienia. Poszczególne dokumenty traktowane są jako zbiory tokenów (słów), a więc pomijana jest całkowicie kolejność słów, powiązania między nimi oraz struktura/gramatyka zdania. Alternatywnie moglibyśmy uwzględnić kolejność biorąc pod uwagę zbiory składające się z 2 słów (bigramy) czy 3 słów (trigramy). Ogólnie taka analiza nosi nazwę modelu n-gram, gdzie n oznacza na jak długich ciągach kolejnych słów chcemy się skupić.

W powyższym przykładzie liczby reprezentujące tekst to liczby wystąpień poszczególnych słów w tekście. Jeszcze prostsza odmiana tej reprezentacji tekstu zakłada, że zamiast wystąpień w macierzy umieszczane są jedynie wartości 0, jeśli dane słowo nie wystąpiło w dokumencie, a 1 — jeśli wystąpiło (niezależnie od liczby wystąpień). Taka reprezentacja nosi nazwę reprezentacji boolowskiej (Cieślewicz, Pelikant, 2009). Aby ją uzyskać, wystarczy podać przy wywołaniu *CountVectorizer* parametr `binary = True` (domyślna jego wartość to *False* — dlatego domyślnie otrzymujemy reprezentację ilościową). Uzyskana tabela zawiera jedynki i zera stanowiące boolowską reprezentację zbioru analizowanych dokumentów.

```
>>> vectorizer = CountVectorizer(binary = True)
>>> X = vectorizer.fit_transform(docs)
>>> print(vectorizer.get_feature_names())
['alligators', 'alpacas', 'and', 'ants', 'five', 'three',
'two']
>>> print(X.toarray())
[[1 1 1 0 0 1 1]
 [1 1 1 1 0 0 1]
 [0 0 0 1 1 0 0]]
```

Prześledźmy jeszcze jeden przykład, przy czym tym razem przy inicjowaniu *CountVectorizer* podamy parametr `ngram_range(2, 2)`, który informuje, że będziemy analizować tylko bigramy.

```
>>> vectorizer = CountVectorizer(ngram_range = (2,2))
>>> X = vectorizer.fit_transform(docs)
>>> print(vectorizer.get_feature_names())
['alligators two', 'alpacas and', 'and three', 'and two',
'five ants', 'three alligators', 'two alligators', 'two al-
pacas', 'two ants']
>>> print(X.toarray())
[[1 1 1 0 0 1 1 1 0]
```

```
[1 1 0 1 0 0 1 1 1]
[0 0 0 0 1 0 0 0 0]]
```

Bigramy zostały wypisane w kolejności alfabetycznej. Prześledźmy ponownie środkowy wiersz tabeli. Wartości 1 w nim zapisane odpowiadają następującym bigramom: „*alligators two*", „*alpacas and*", „*and two*", „*two alligators*", „*two alpacas*", „*two ants*", a zera pozostałym bigramom. Wynika stąd, że wyjściowe zdanie „*two alligators, two alpacas and two ants*" zostało podzielone w sposób przedstawiony w tabeli:

„two"	„alligators"	„two"	„alpacas"	„and"	„two"	„ants"
Bigram 1						
	Bigram 2					
		Bigram 3				
			Bigram 4			
				Bigram 5		
					Bigram 6	

Moglibyśmy użyć jednocześnie uni- oraz bigramów podając ngram_range = (1, 2). W praktyce wybór n w dużej mierze zależy kontekstu. Dobrą praktyką jest eksperymentowanie z różnymi wartościami i obserwowanie zachowania modeli uczenia maszynowego.

Inny sposób reprezentacji tekstu nosi nazwę TF-IDF (TF — *term frequency*, IDF — *inverse document frequency*). Tak jak w poprzednim przykładzie, następuje tutaj konwersja tekstu na macierz, w której każdy wiersz reprezentuje dokument, a każda kolumna reprezentuje słowo. Różnica polega na tym, że poszczególne elementy macierzy odpowiadają znaczeniu (wadze) konkretnego słowa w danym dokumencie, które wyraża następujący wzór:

$$w_{i,j} = tf_{i,j} * ln\left(\frac{N}{df_i}\right),$$

gdzie $w_{i,j}$ oznacza komórkę macierzy, w której wiersze (i) oznaczają poszczególne dokumenty, a kolumny (j) to analizowane n-gramy (pamiętamy, że mogą to być unigramy, czyli poszczególne słowa). Pierwszy czynnik iloczynu ($tf_{i,j}$ — *term frequency*) oznacza częstość względną, czyli stosunek liczby wystąpień danego n-gramu j w dokumencie i do sumy liczby wystąpień wszystkich n-gramów (uwzględnionych w analizie) w tym dokumencie. Drugi czynnik $ln(N/df_i)$ wyraża IDF, czyli odwrotną częstość dokumentu obliczaną jako odwrócony iloraz liczby dokumentów, w których pojawia się dane słowo j (df_i) do

liczby wszystkich analizowanych dokumentów (*N*). Zabieg ten ma na celu wychwycenie słów, które są charakterystyczne dla poszczególnych dokumentów, tzn. występują jedynie w kilku dokumentach w korpusie.

```
>>> from sklearn.feature_extraction.text import TfidfVecto-
rizer
>>> vectorizer = TfidfVectorizer()
>>> X = vectorizer.fit_transform(docs)
>>> print(vectorizer.get_feature_names())
['alligators', 'alpacas', 'and', 'ants', 'five', 'three',
'two']
>>> print(X.toarray())
[[ 0.58398432  0.29199216  0.29199216  0.          0.
0.3839346
   0.58398432]
 [ 0.2773501   0.2773501   0.2773501   0.2773501   0.
0.
   0.83205029]
 [ 0.          0.          0.          0.60534851
0.79596054  0.          0.        ]]
```

W pakiecie *sklearn.feature_extraction.text* obliczanie poszczególnych elementów macierzy odbywa się w nieco zmodyfikowany sposób[6].

Po pierwsze, *tf* jest tutaj liczbą wystąpień danego n-gramu w dokumencie (czyli wynikową działania *CountVectorizer*). Po drugie, IDF zdefiniowana jest jako $ln\left(\frac{1+N}{1+df_i}\right)+1$.

Obliczone iloczyny TF-IDF są następnie znormalizowane — tzn. podzielone przez długość wektora odpowiadającego danemu dokumentowi w przestrzeni euklidesowej.

Prześledźmy obliczenia na przykładzie. Z tabeli powyżej widzimy, że wartość odpowiadająca słowu „*alligators*" w doc1 wynosi 0.58398432. Wielkość ta obliczona została jako wynik mnożenia 2 (liczba wystąpień „*alligators*" w doc1) oraz ln((1+3)/(1+2))+1. Przypomnijmy, że *N* — liczba dokumentów w korpusie w naszym wypadku wynosi 3, natomiast liczba dokumentów, w których wystąpiło analizowane słowo to 2. Otrzymany wynik to 2.57536414. Wyniki tych obliczeń powtórzonych dla kolejnych słów w doc1 przedstawiliśmy w tabeli poniżej.

	„alligators"	„alpacas"	„and"	„ants"	„five"	„three"	„two"
doc1	2.57536414	1.28768207	1.28768207	0.00000000	0.00000000	1.69314718	2.57536414

[6] Poniżej omawiamy wywołanie narzędzia z opcjami domyślnymi. Do szerszych informacji odsyłamy Czytelnika do dokumentacji.

Formuła umożliwiająca obliczenie ostatniego elementu (wartości odpowiadającej słowu „*two*") to 2*(ln((1+3)/(1+2))+1).

Wartości te należy teraz znormalizować dzieląc je przez długość wektora w przestrzeni euklidesowej, która w analizowanym przypadku wynosi

$$\sqrt{\left(\begin{matrix}2.57536414^2+1.28768207^2+1.28768207^2+1.69314718^2+\\+2.57536414^2\end{matrix}\right)}\approx 4.41.$$

Dzieląc wartości poszczególnych komórek tabeli przez tę liczbę uzyskamy wartości z pierwszego wiersza w tabeli X.

Spójrzmy jeszcze raz na pełną tabelę pokazującą wartości TF-IDF dla naszego zbioru tekstów w bardziej czytelnej formie:

	„alligators"	„alpacas"	„and"	„ants"	„five"	„three"	„two"
doc1	0.58398432	0.29199216	0.29199216	0	0	0.38393460	0.58398432
doc2	0.27735010	0.27735010	0.27735010	0.27735010	0	0	0.83205029
doc3	0	0	0	0.60534851	0.79596054	0	0

Wynika z niej, że największą wagę otrzymało słowo „*two*" w doc2, które wystąpiło w nim trzykrotnie oraz dwukrotnie w doc1. Nieco niższą wartość otrzymało słowo „*five*" w doc3, które wystąpiło jedynie w tym dokumencie. Słowa, które wystąpiły jednokrotnie w 2 z 3 analizowanych tekstów („*alpacas*", „*and*", „*ants*") otrzymały dodatnie wagi rzędu 0.277–0.605[7]. W tym przypadku różnice pomiędzy poszczególnymi wartościami wynikają z długości wektorów.

Ogólnie, największe znaczenie dla klasyfikacji teksów mają słowa „ze środka rozkładu", tzn. występujące niezbyt często, ponieważ słowa występujące we wszystkich tekstach jednocześnie nie będą cechą wyróżniającą, z drugiej zaś strony — słowa uzyskujące wysokie wagi nie mogą występować zbyt rzadko, ponieważ nie wnoszą wówczas informacji o różnicach między tekstami.

Słowa, które występują szczególnie często i/lub nie mają znaczenia klasyfikacji dokumentu noszą nazwę *stopwords* (stoplisty). Są to na przykład słowa typu „*że*", „*oraz*", „*więc*", jednak co do zasady lista *stopwords* jest zależna od kontekstu badania. Na przykład gdybyśmy chcieli zidentyfikować wśród ogółu pozwów sądowych tylko te, które dotyczą dziedziczenia, prawdopodobnie słowa typu „*sąd*", „*powód*", czy np. nazwy poszczególnych miesięcy, nie byłyby dobrymi cechami różnicującymi, ponieważ występowałyby w (prawie) wszystkich badanych dokumentach. Prawdopodobnie cechą różnicującą byłoby słowa typu: „*spadek*", „*testament*", „*zmarły*", „*dziedziczyć*", „*spadkobierca*" itp., a także np. wartość przedmiotu sporu, którą moglibyśmy uzyskać poprzez użycie wyrażeń regularnych opisanych w rozdziale 4.2.

[7] W przypadku jeśli słowo nie występuje w danym dokumencie, waga wynosi 0.

Istnieją jednak dość uniwersalne listy *stopwords* dla poszczególnych języków. Przykład takiej listy jest dostępny poprzez pakiet *nltk*:

```
>>> from nltk.corpus import stopwords
>>> print(stopwords.words('english'))
```

W celu ograniczenia wolumenu analizowanych danych zazwyczaj elementem wstępnego przetworzenia tekstu jest usunięcie słów ze stoplisty. Innymi słowy stoplista jest zdefiniowana przed rozpoczęciem właściwej analizy. Alternatywnym rozwiązaniem jest wybór słów (ogólnie n-gramów) podlegających dalszej analizie, dokonany na podstawie częstości ich występowania (por. Manning, Raghavan, Schütze, 2009, s. 27).

Warto dodać, że dodatkowymi cechami tekstu (oprócz tych wynikających z zaprezentowanej wyżej ich reprezentacji) mogą być np. częstość wykorzystywania znaków interpunkcyjnych, liczba słów zapisanych wielką literą czy np. długość słów.

5. Uczenie maszynowe

5.1 Uczenie nadzorowane i klasyfikacja — informacje wstępne

Rozdział 5 jest poświęcony metodom **uczenia maszynowego**. Zanim opiszemy dwa najważniejsze problemy uczenia (klasyfikację i regresję), wyjaśnimy Czytelnikowi co kryje się pod pojęciem uczenia maszynowego. Tradycyjne metody przetwarzania danych (w tym procesy biznesowe — typu księgowanie, zatwierdzenie płatności, tworzenie raportów, wypełnianie deklaracji podatkowych itp.) polegają na regułach. Reguły te są określone przez przepisy lub ekspertów, a w połączeniu z danymi wejściowymi uzyskujemy wynik. Na przykład określone wydatki są księgowane jako koszty uzyskania przychodu. Decyduje o tym księgowy, który rozumie odpowiednie przepisy (reguły). Analityk korzystający z metod uczenia maszynowego rozwiązałby to inaczej. Zgromadziłby duży zbiór (tysiące, jeśli nie setki tysięcy) dotychczasowych księgowań, który zawierałby zarówno dane wejściowe (faktury i inne dokumenty źródłowe), jak i prawidłowe zapisy księgowe. W podejściu **uczenia maszynowego**[1] zestaw danych wejściowych w połączeniu z wynikiem pozwala na znalezienie reguły.

Klasyfikacja polega na znalezieniu algorytmu (reguły) pozwalającej na rozróżnienie pomiędzy skończoną liczbą kategorii. Wracając do przykładu z księgowaniem, moglibyśmy wyróżnić księgowanie po stronie kosztów lub przychodów albo księgowanie np. na konto[2] 100 czy 130. Metody te, podobnie jak regresja (rozdz. 5.8), w której poszukujemy reguły określającej war-

[1] Przynajmniej jeśli chodzi o uczenie nadzorowane i uczenie ze wzmocnieniem.

[2] Czytelnikom słabiej zaznajomionych z zasadami rachunkowości wyjaśnimy, że zwyczajowo konto 100 przeznaczone jest do księgowania operacji gotówkowych („kasa"), a 130 — do wszystkich operacji z rachunku bieżacego.

tość zmiennej wyjściowej, zaliczają się metod uczenia nadzorowanego. Mimo, że w książce opisujemy wyłącznie metody uczenia nadzorowanego, przedstawimy krótko Czytelnikowi trzy najważniejsze rodzaje (bardzo ogólne) uczenia maszynowego (np. Rashka, 2015, s. 2, Bengio i in., 2018, s. 102–104, Chinnamgari, 2019).

O **uczeniu nadzorowanym** mówimy w przypadku gdy posiadamy zbiór z „prawidłowymi'' wartościami zmiennej wyjściowej. Jeśli jako przykład uczenia maszynowego weźmiemy rozpoznawanie obrazów (równie dobrze: transakcji, faktur, klientów) to korzystamy ze zbioru uczącego, w którym oprócz samego obrazu ktoś nam podał np. czy dany obraz przedstawia roślinę czy zwierzę (klasyfikacja) albo ile lat ma człowiek przedstawiony na zdjęciu (regresja). Na podstawie tej próby algorytm uczenia maszynowego jest w stanie nauczyć się rozróżnić (klasyfikować) roślinę od zwierzęcia albo szacować wiek ludzki na podstawie zdjęć. Mówimy także, że jest to uczenie „z nauczycielem''.

W przypadku **uczenia nienadzorowanego** nie posiadamy „prawidłowej odpowiedzi''. Widzimy po prostu zdjęcia i możemy podzielić je np. na dwie grupy „różniące się''. Może się zdarzyć, że kryterium podziału będzie pomiędzy rośliną a zwierzęciem, ale równie dobrze możemy pogrupować obrazy np. na czarno-białe i kolorowe.

Jeszcze innym sposobem uczenia maszynowego jest **uczenie przez wzmacnianie** (*reinforcment learning*). Jest to uczenie metodą prób i błędów, gdzie nie mamy jawnie podanej funkcji-kryterium, według której oceniamy jakość modelu. Uczenie przebiega metodą prób i błędów, a po każdej próbie dostajemy informację zwrotną. Odpowiada to grze komputerowej, gdzie gracz steruje np. zachowaniem postaci czy pojazdu, a po zakończonej grze dostaje informację o liczbie punktów. Dzięki temu, powtarzając grę wiele razy, jesteśmy w stanie nauczyć się osiągać dobre wyniki w grze, nawet gdy zasady gry nie są podane wprost.

5.1.1 Jak przebiega uczenie nadzorowane

Celem budowy modelu uczenia maszynowego jest znalezienie zależności pomiędzy zmiennymi wejściowymi (inaczej objaśniającymi) a zmienną wyjściową (objaśnianą). Zwykle przyjmuje się, że zmienne wejściowe są przyczynami wobec zmiennej wyjściowej[3].

Poszukiwana zależność jest ustalana na podstawie próbki — dzięki znajomości wzorca. Jest to także nazywane uczeniem „z nauczycielem'', gdyż model „uczy się'' na podanych wzorcach. Takim wzorcem jest próba, czyli zestaw

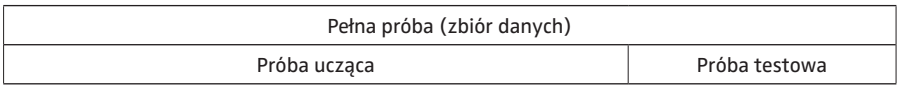
[3] Jest to regułą, lub przynajmniej pożądaną własnością. W modelach uczenia maszynowego wymóg ten nie jest jednak ściśle przestrzegany, zwłaszcza że interpretacja otrzymanej zależności ma mniejsze znaczenie niż zdolność modelu do generowania trafnych prognoz. Szersza dyskusja różnic w tych podejściach przedstawiona jest w pracy Shmueli (2010).

obserwacji zmiennej objaśnianej wraz z cechami je opisującymi (zmiennymi objaśniającymi).

W rozdziale przede wszystkim rozważamy klasyfikację binarną, gdzie zmienna objaśniana przyjmuje wartości 0 lub 1, zaś w przypadku regresji zmienna objaśniana przyjmuje w zasadzie dowolne wartości.

Używając dużej liczby zmiennych wejściowych możemy znaleźć model, który na zasadzie przypadku skutecznie klasyfikuje badaną zmienną (przeuczenie, dopasowanie się do szumu), dlatego dzielimy próbę na próbę treningową i testową (Rys. 5.1).

Rys. 5.1 Podział próby



Co oznacza wspomniane wyżej przeuczenie i jak podział próby zapobiega temu niepożądanemu zjawisku? Wyobraźmy sobie cały proces klasyfikacji analogicznie do nauczania w szkole i odpowiedzi w teście jednokrotnego wyboru. Oczywiście w procesie (*nomen omen*) nauczania nie chodzi o to, żeby nauczyć się „na pamięć'' zestawu zadań ćwiczonych w trakcie zajęć. Tak naprawdę celem jest nauczenie się wszystkich zadań, szczególnie tych, których się jeszcze nie widziało.

Próba treningowa — odpowiednik „lekcji w klasie'', próba testowa — odpowiednik klasówki (stąd cross-walidacja byłaby metodą wystawienia oceny na koniec roku).

Po omówieniu ogólnych zasad klasyfikacji, w kolejnym podrozdziale przedstawimy wybrane metody klasyfikacji. Za każdym razem zakładamy, że Czytelnik posiada zainstalowane (i zaimportowane) biblioteki: *pandas* oraz *sklearn*[4]. W poniższych przykładach będziemy wykorzystywać zbiór danych z dziedziny diagnostyki medycznej, gdzie celem jest określenie czy zmiana jest złośliwa (0) czy łagodna (1) (zob. https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))

```
>>> from sklearn.datasets import load_breast_cancer
>>> data = load_breast_cancer()
>>> df_dane = pd.DataFrame(data.data, columns=data.featu-
re_names)
```

Zgodnie wyżej wymienionymi regułami weryfikacji modeli uczenia maszynowego, podzielimy ten zbiór na próbę treningową i testową (odpowiednio 430 obserwacji początkowych i 139 ostatnich obserwacji, tj. ok. 75% zbioru przypadnie na zbiór treningowy).

[4] Pełna nazwa tej biblioteki to Scikit-learn (ale w kodzie Python używamy tylko nazwy *sklearn*).

```
>>> df_tren = df_dane[0:430]
>>> df_test = df_dane[430:]
>>> y_tren = data.target[0:430]
>>> y_test = data.target[430:]
```

W praktyce taki podział według indeksu zgodnego z kolejnością w arkuszu czy bazie danych nie zawsze jest najlepszy (np. gdy oryginalne dane posortowano według wartości zmiennej objaśnianej, wówczas struktura próby testowej będzie się drastycznie różniła od struktury próby treningowej), możemy wykonać podział losowy. Biblioteka *sklearn* dostarcza nam w tym celu gotowego i prostego do użycia narzędzia:

```
>>> from sklearn.model_selection import train_test_split
>>> X = data.data
>>> y = data.target
>>> X_tren, X_test, y_tren, y_test = train_test_split(X, y,
test_size=0.25)
```

Oczywiście, nic nie stoi na przeszkodzie, żeby po niewielkich modyfikacjach zastosować te przykłady dla innych zbiorów danych, szczególnie dla zbiorów, którymi posługujemy się w pracy.

5.1.2 Klasyfikacja binarna

Klasyfikacja zakłada skończoną liczbę kategorii — możliwych wartości zmiennej objaśnianej (zwykle nie przekraczającą kilkunastu), ale my będziemy rozpatrywać przede wszystkim klasyfikację binarną. W ostatnim rozdziale przedstawimy możliwości uogólnienia metod na więcej niż dwie kategorie. Wracając jeszcze na chwilę, spójrzmy na problem studenta przystępującego do egzaminu. Jeśli jesteśmy zainteresowani przewidywaniem czy student zda egzamin czy nie, wówczas mówimy o klasyfikacji binarnej. Z kolei jeśli chcemy przewidzieć ocenę z egzaminu (np. w skali: 2, 3, 4 lub 5), wówczas jest to zagadnienie klasyfikacji wieloklasowej. Wreszcie, jeśli interesuje nas dokładna liczba punktów z egzaminu, wówczas mówimy o regresji[5].

Wśród wielu praktycznych zastosowań klasyfikacji w ekonomii i zarządzaniu możemy wymienić (zob. Finlay, 2010 oraz literatura cytowana poniżej):

– przewidywanie zachowania użytkownika serwisu internetowego (np. czy użytkownik odpowie na reklamę, doda towar do koszyka, zakończy zakupy itp.),

[5] Uważny Czytelnik mógłby zauważyć, że liczba punktów jest całkowita, dlatego jest to problem klasyfikacji z wieloma kategoriami. Ściśle rzecz ujmując to prawda, jednak gdy zmienna dyskretna posiada więcej niż 9 klas (kategorii) i możemy wskazać jednoznacznie wartości „większe" i „mniejsze", wówczas regresja jest dobrym przybliżeniem, a nawet jest zalecana (np. Gruszczyński, 2012; Train, 2009).

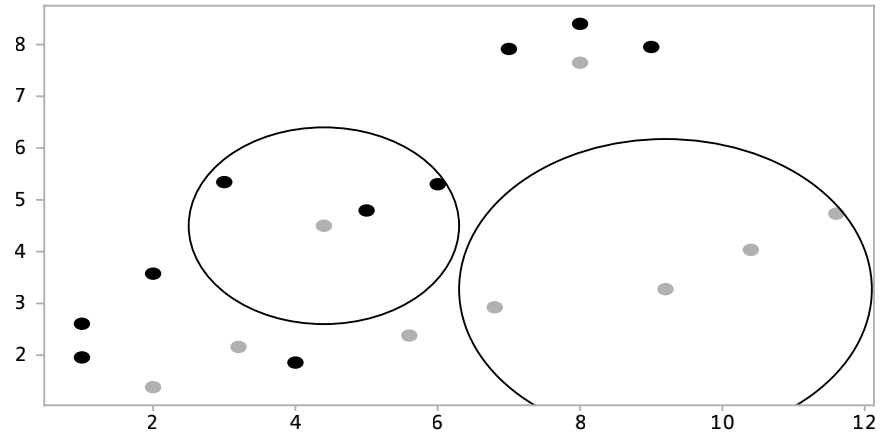
- wykrywanie nieuczciwych klientów (oszustw, prania brudnych pieniędzy itp.),
- analizę wystąpienia szkody ubezpieczeniowej,
- przewidywanie upadłości przedsiębiorstw lub niewypłacalności klientów banku,
- wskazywanie klientów zainteresowanych danym produktem lub promocją,
- analizę ryzyka utraty (odejścia) klienta (Verbeke i in., 2011),
- wykrywanie niechcianych wiadomości — tzw. spamu (Crawford i in., 2015),
- rozpoznawanie cyberataków na serwery i inne systemy komputerowe (Buczak, Guven, 2015),
- czy druga strona transakcji wypowie się pozytywnie lub negatywnie?
- określenie czy dany tekst (np. artykuł prasowy) dotyczy określonego zagadnienia,
- określenie czy dany tekst ma pozytywny lub negatywny wydźwięk emocjonalny,
- określenie jaki rodzaj towaru lub usługi wybierze klient (np. czy kupi produkt droższy czy tańszy).

5.2 Metoda najbliższych sąsiadów

W metodzie najbliższych sąsiadów (KNN) przewidujemy na podstawie *K* obserwacji leżących „najbliżej" analizowanego punktu. Kryterium bliskości jest określone w przestrzeni zmiennych wejściowych, na bazie standardowej odległości euklidesowej[6]. Dla przykładu 2 zmiennych wejściowych moglibyśmy rysować okręgi o środkowym punkcie analizowanym, zwiększając średnicę aż do momentu gdy wewnątrz okręgu znajdzie się dokładnie *K* „sąsiadów". Ilustrację graficzną przedstawiamy na Rys. 5.2 (każda obserwacja to jeden punkt, stanowiący parę wartości dwu zmiennych wejściowych; kolorem oznaczono wartości zmiennej wyjściowej — czarny to y=1, a szary y=0; dodatkowo na wykresach punkty y=1 są oznaczone większym znacznikiem lub grubszą linią). W modelu najbliższych sąsiadów obserwację klasyfikujemy jako 1 jeśli większość obserwacji w tak rozumianym otoczeniu była 1 (zob. Rys. 5.3). W przeciwnym przypadku, gdy większość obserwacji w otoczeniu była 0, daną obserwację klasyfikować będziemy jako 0. Tak więc metoda ta działa na zasadzie „głosowania" („głosuje" każdy punkt sąsiedni) czy dany punkt jest czarny czy szary. Jeśli większość punktów w sąsiedztwie stanowią punkty czarne (szare) dany punkt jest zaklasyfikowany jako czarny (szary). Przyjęcie jako *K* liczby nieparzystej eliminuje możliwość uzyskania remisu (np. dla *K*=6, gdyby w sąsiedztwie znalazły się dokładnie 3 punkty czarne i 3 szare) i niejednoznaczność z tego wynikającą.

[6] Odpowiada to odległości „w linii prostej" łączącej dwa punkty.

Rys. 5.2 Klasyfikacja przy pomocy najbliższych sąsiadów (*K*=3, dwie zmienne wejściowe wartości X1 na osi poziomej, a X2 na pionowej)

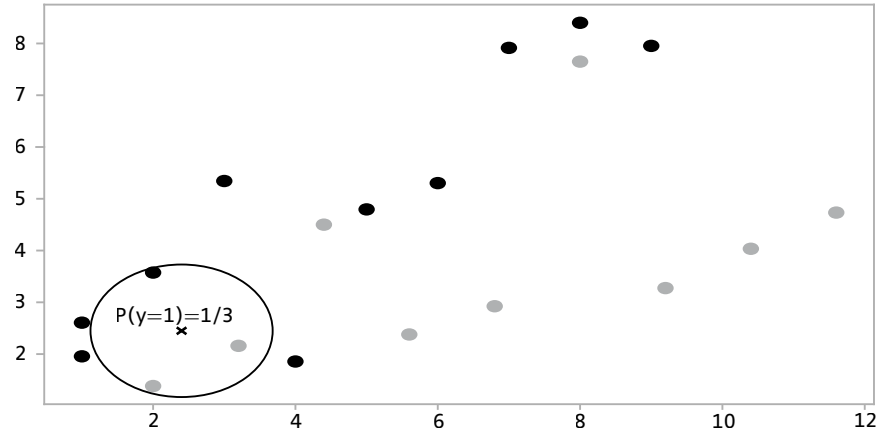


Inicjalizacja obiektów-modeli uczenia maszynowego:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn_3 = KNeighborsClassifier(n_neighbors=3)
```

Jak widzimy powyżej, inicjalizacja obiektu wymaga podania liczby sąsiadów (*K*). Sam dobór tego parametru warunkuje pożądane własności klasyfikatora (im większe *K*, tym mniejsza szansa na przeuczenie, ale tym gorsze dopasowanie w przypadku gdy zależność jest bardziej złożona, np. silnie nieliniowa; zob. Li, Kurita, 2015; James i in. 2017, s. 151–154). Wydaje się, że najlepiej kwestię tę można rozstrzygnąć na bazie porównania jakości klasyfikacji (rozdz. 5.6).

Rys. 5.3 Predykcja punktu spoza zbioru uczącego (nowy pkt oznaczony symbolem „x")



Dla modelu *K*=3 najbliższych sąsiadów uczenie modelu w oparciu o próbę uczącą wymaga wywołania metody **.fit()** z podaniem zmiennych wejściowych (`df_tren`) oraz zmiennej wyjściowej (`y_tren`):

```
>>> knn_3.fit(df_tren, y_tren)
>>> yknn_pred_tren = knn_3.predict(df_tren)
>>> yknn_pred_test = knn_3.predict(df_test)
```

Jak zobaczymy, działanie metod **.fit()** oraz **.predict()** jest identyczne we wszystkich modelach z modułu *sklearn*. Z tego względu kody ulegną minimalnej zmianie i nie będziemy ich ponownie omawiać. Skoncentrujemy się za to na opisie sposobu działania metod statystycznych.

W wielu przypadkach zależy nam nie tylko na klasyfikacji 0 lub 1, ale chcemy oszacować prawdopodobieństwo z jakim dana obserwacja będzie zaklasyfikowana jako 1 (bądź 0). W metodzie najbliższych sąsiadów prawdopodobieństwo to oblicza się jako odsetek (udział) obserwacji z danej kategorii w otoczeniu punktu[7]. Przykładowo, w przypadku zobrazowanym na Rys. 5.3 w otoczeniu punktu „x'' znajduje się tylko jedna obserwacja y=1 (etykieta „kolor czarny''), stąd jak widać prawdopodbieństwo z jakim „x'' miałby etykietę „kolor czarny'' wynosi 1/3. Jeśli chodzi o ujęcie w Python, służy do tego metoda **.predict_proba()**.

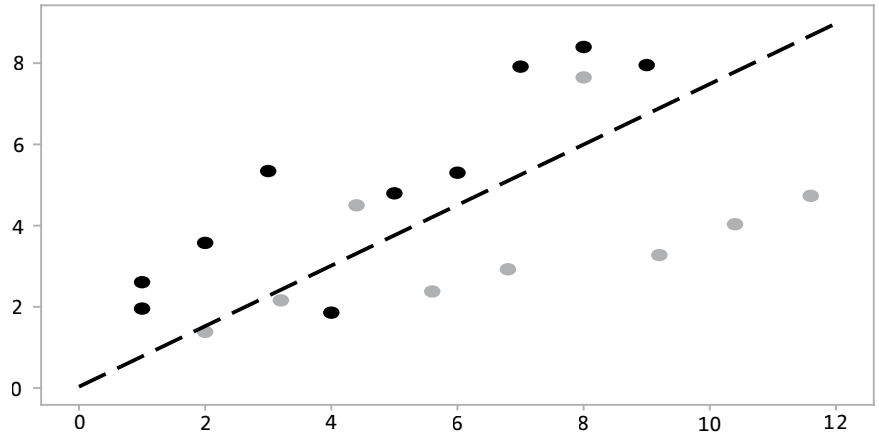
5.3 Liniowa analiza dyskryminacyjna

O ile w metodzie najbliższych sąsiadów klasyfikujemy na podstawie podobieństw, to liniowa analiza dyskryminacyjna poszukuje hiperpłaszczyzny oddzielającej dwie kategorie w ten sposób, że po jednej stronie hiperpłaszczyzny znajdą się głównie obserwacje 1, a po drugiej — głównie 0. W związku z tym ta hiperpłaszczyzna staje się pewnego rodzaju „barierą'' oddzielającą poszczególne kategorie. Aby przybliżyć pojęcie hiperpłaszczyzny powiedzmy, że w przestrzeni dwu zmiennych poszukujemy linii prostej najlepiej oddzielającej obserwacje z obu klas (Rys. 5.4), a w przestrzeni trójwymiarowej byłaby to płaszczyzna[8]. Dla ilustracji przypadku z dwoma zmiennymi wejściowymi wykorzystujemy te same dane, co w rozdziale 5.2.

[7] Dlatego przy klasyfikacji binarnej metodą *K* najbliższych sąsiadów zaleca się *K* nieparzyste. Eliminuje to trudną do interpretacji sytuację, gdy zarówno prawdopodobieństwo, że y=0 jak i y=1 jest równe i wynosi ½.

[8] W bardziej ogólnym przypadku gdzie mamy *M* zmiennych wejściowych, będzie to *M-1* wymiarowa hiperpłaszczyzna (zob. James i in., 2017, s. 338–341).

Rys. 5.4 Klasyfikacja przy pomocy liniowej funkcji dyskryminacyjnej (dwie zmienne wejściowe)



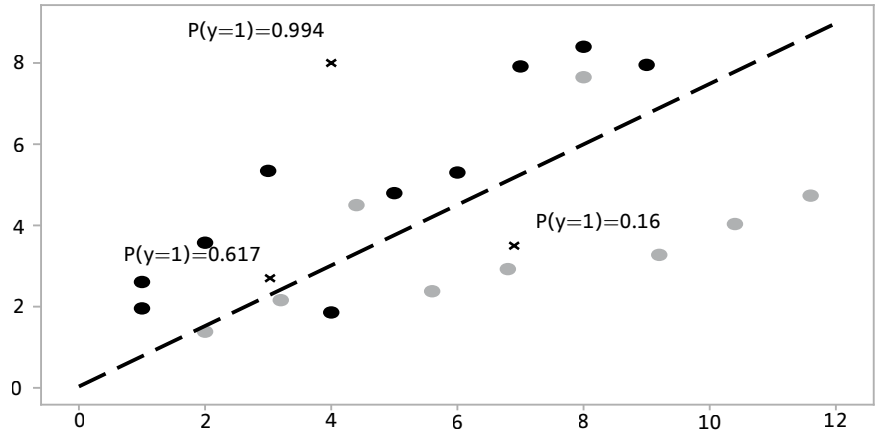
W przeciwieństwie do KNN nie wymaga się podania żadnych parametrów.

```
>>> from sklearn.discriminant_analysis import LinearDi-
scriminantAnalysis
>>> lda = LinearDiscriminantAnalysis()
```

Dalsza część analiz przebiega identycznie jak w przypadku wcześniej omówionego klasyfikatora KNN:

```
>>> lda.fit(df_tren, y_tren)
>>> y_pred_train = lda.predict(df_train)
>>> y_pred_test = lda.predict(df_test)
```

Rys. 5.5 Predykcja prawdopodobieństwa, że dana obserwacja posiada etykietę 1 („kolor czarny")



5.4 Prosty klasyfikator bayesowski

Poprzednio omówiony klasyfikator liniowy akcentował różnice położenia obserwacji 0 i 1 w przestrzeni zmiennych wejściowych (objaśniających). Prosty klasyfikator bayesowski (zwany także naiwnym klasyfikatorem bayesowskim) bada różnice rozkładów prawdopodobieństw zmiennych wejściowych w grupie obserwacji 0 i 1, a następnie łączy to z wiedzą *a priori* (czyli prawdopodobieństwem zaobserwowania 0 i 1, niezależnie od wartości zmiennych wejściowych). Różnice w rozkładach badane są dla każdej zmiennej osobno, a łączne prawdopodobieństwo przynależności obserwacji do danej grupy jest wyliczane przy założeniu niezależności (dodatkowo, w celu obliczenia prawdopodobieństwa konieczne jest założenie postaci rozkładu). Dzięki temu łączne prawdopodobieństwo wystąpienia określonych wartości zmiennych wejściowych wyraża się jako iloczyn prawdopodobieństw dla poszczególnych zmiennych.

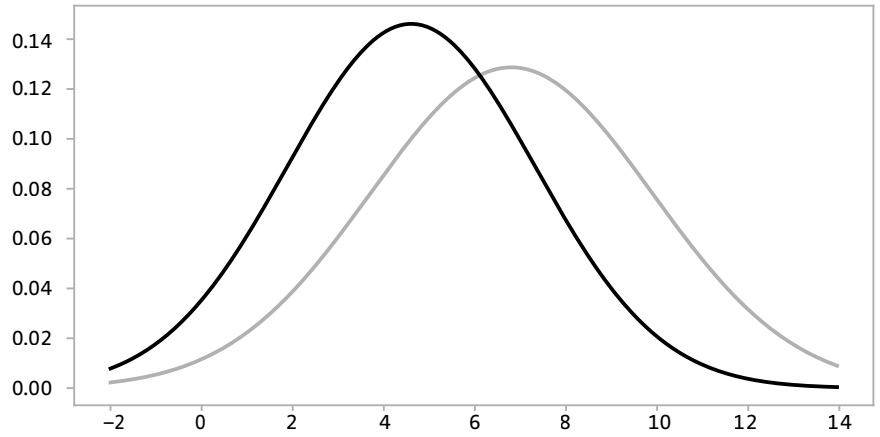
Spróbujmy przedstawić przykład dla „sztucznych" danych z przykładu z rozdziału 5.2. Standardowo dla zmiennych ciągłych zakładamy, że zmienne mają rozkład normalny. Dla rozkładu normalnego podanie średniej oraz odchylenia standardowego wystarcza do określenia całego kształtu rozkładu. Tak więc dla y=1 (czarne punkty na Rys. 5.2):

– średnia wartość zmiennej X1 wynosi 4,6 a odchylenie standardowe 2,73,
– średnia wartość zmiennej X2 wynosi 4,97 a odchylenie standardowe 2,36,

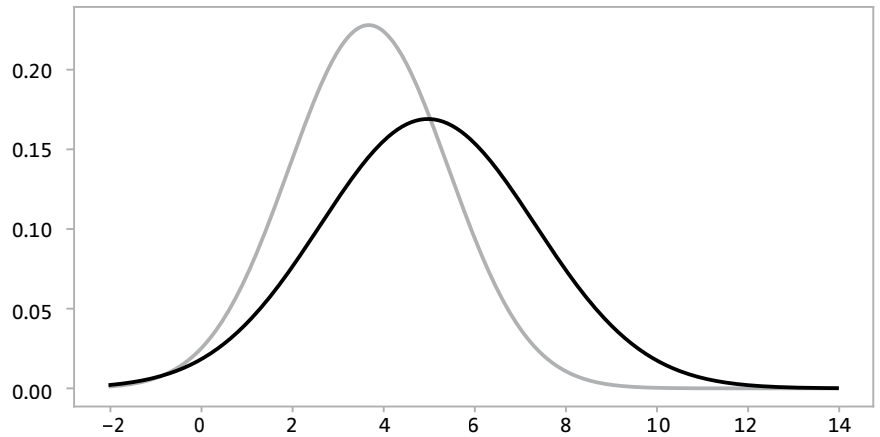
Natomiast dla y=0 (szare punkty na Rys. 5.2):

– średnia wartość zmiennej X1 wynosi 6,8 a odchylenie standardowe 3,1,
– średnia wartość zmiennej X2 wynosi 3,67 a odchylenie standardowe 1,75,

Rys. 5.6 Rozkłady gęstości zmiennej X1 (0 i 1, oznaczenia jak poprzednio kolorami)



Rys. 5.7 Rozkłady gęstości zmiennej X2 (0 i 1, oznaczenia jak poprzednio kolorami)



Zastanówmy się jakie jest prawdopodobieństwo, że obserwacja o wartości X1=2 i X2=3,5 będzie miała etykietę czarną (y=1). Prawdopodobieństwo *a priori*, czyli że zaobserwujemy obserwację y=1 (czyli etykietę oznaczoną kolorem czarnym) wynosi 10/19 tj. ok. 0,53 (10 obserwacji y=1 na 19 obserwacji w zbiorze danych). Jak wspominaliśmy, w klasyfikatorze Bayesa każdą zmienną traktujemy osobno. Dla X1=2 gęstość rozkładu normalnego dla zbioru obserwacji y=1 wynosi 0,093, a dla y=0 wynosi 0,039[9], co oznacza, że taka wartość zmiennej X1 jest bardziej prawdopodobna dla obserwacji y=1. Z kolei

[9] Przybliżone wartości gęstości możemy odczytać graficznie z wykresów 5.6 (dla zmiennej X1) i 5.7 (dla X2). W celu dokładnego obliczenia tych wielkości można użyć np. z funkcji **norm()** z modułu *scipy.stats*.

wartość X2=3,5 odpowiada gęstości rozkładu wynoszącą: 0,139 (obserwacje y=1) i 0,227 (y=0). Zestawiając podane gęstości z prawdopodobieństwem *a priori* otrzymamy interesujący nas wynik[10] — dla podanych wartości X1 i X2 prawdopodobieństwo, że obserwacja będzie posiadała etykietę czarną (y=1) wynosi ok. 0,62.

Oczywiście powyższy przykład w pakiecie *sklearn* wymagałby dużo mniej wysiłku, bowiem wszystkie te czynności wykonują za nas standardowe metody **.fit()**, **.predict()** oraz **.predict_proba()**.

```
>>> from sklearn.naive_bayes import GaussianNB
>>> bayes = GaussianNB()
>>> bayes.fit(df_tren, y_tren)
>>> y_pred_train = bayes.predict(df_tren)
>>> y_pred_test = bayes.predict(df_test)
```

5.5 Regresja logistyczna

W regresji logistycznej poszukujemy parametrów najlepiej opisujących zależności pomiędzy zmiennymi wejściowymi a binarną zmienną wyjściową. Koncepcja ta zbliżona jest to regresji liniowej (rozdz. 5.8), lecz zamiast zależności liniowej poszukujemy zależności nieliniowej, ale monotonicznej (określonej przez dystrybuantę rozkładu logistycznego).

Estymacja parametrów przebiega przy pomocy metody największej wiarygodności. Podobnie jak poprzednio, skoncentrujemy się na przedstawieniu implementacji w Pythonie, natomiast szczegóły estymacji zawarte są np. w: Gruszczyński (2012) czy Train (2009).

```
>>> from sklearn.linear_model import LogisticRegression
>>> logit = LogisticRegression()
```

Estymacja parametrów modelu regresji logistycznej przebiega przy pomocy metody **.fit()**, analogicznie do poprzednio omówionych metod.

```
>>> logit.fit(df_tren, y_tren)
>>> y_pred_train = bayes.predict(df_tren)
>>> y_pred_test = bayes.predict(df_test)
```

5.6 Miary jakości klasyfikacji i porównanie klasyfikatorów

Jak wiemy zmienna wyjściowa przyjmuje dwie wartości (0 i 1), podobnie jak rezultaty prognoz. Dlatego pełną informację o rozkładzie błędów daje macierz 2x2, w wierszach rzeczywiste wartości (obserwowane), a w kolumnach warto-

[10] Oszczędzimy Czytelnikowi wzór i szczegóły obliczeń. Zainteresowanych szczegółami odsyłamy np. do James i in., 2017, s. 139.

ści przewidywane przez model (prognozowane). W takim ujęciu interesuje nas liczba wystąpień każdego z czterech przypadków:

– obserwacja w rzeczywistości y=1 zaklasyfikowana przez model jako 1 (*true positive*, TP),
– obserwacja y=1 zaklasyfikowana jako 0 (*false negative*, FP),
– obserwacja w rzeczywistości y=0 zaklasyfikowana jako 0 (*true negative*, TN),
– obserwacja y=0 zaklasyfikowana jako 1 (*false positive*, FP).

Macierz zawierającą liczbę takich przypadków nazywamy macierzą błędów klasyfikacji (*confusion matrix*). Macierz taką możemy wyznaczyć zliczając liczbę każdego z przypadków przy pomocy operacji logicznych bądź skorzystać z funkcji **confusion_matrix()** z modułu *scikit-learn*.

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_test, yknn_pred_test)
array([[33,  2],
       [ 8, 96]])
```

– Współczynnik trafności (*accuracy*) — definiowany jako odsetek trafnych predykcji (iloraz liczby obserwacji trafnie zaklasyfikowanych do liczby obserwacji),
– Czułość (*recall*) — odsetek trafnie zaklasyfikowanych obserwacji 1 (iloraz liczby trafnie zaklasyfikowanych 1 do liczby obserwacji empirycznych 1),
– Precyzja klasyfikacji negatywnej (swoistość, *specificity*) — odsetek trafnie zaklasyfikowanych obserwacji 0 (iloraz liczby trafnie zaklasyfikowanych 0 do liczby obserwacji empirycznych 0),
– Współczynnik F1 — statystyka przedstawiająca łączną zdolność klasyfikacji obserwacji pozytywnych i negatywnych (zbilansowana miara czułości i precyzji, określających odpowiednio jaki odsetek klasyfikacji 0 i 1 jest zgodne z rzeczywistą wartością zmiennej wyjściowej tj. y); statystyka ta zawiera się w przedziale <0,1> i przyjmuje wartość zero gdy czułość lub precyzja są zerowe.

```
>>> from sklearn.metrics import accuracy_score, f1_score,
recall_score
>>> accuracy_score(y_test, yknn_pred_test)
0.9280575539568345
```

Przy pomocy opisanych wyżej metod możemy wygenerować pojedyncze wskaźniki świadczące o jakości modelu. Jeśli oceniamy model na podstawie wielu wskaźników, moduł *sklearn* oferuje nam funkcję, która zwróci krótkie podsumowanie najistotniejszych wskaźników jakości predykcji — **sklearn.metrics.classification_report()**. Funkcja ta wymaga podania dwu argumentów — rzeczywiste (empiryczne) wartości zmiennej wyjściowej (y) oraz wynik

klasyfikacji z modelu (predykcji). Poniżej przedstawiamy przykład zastosowania tej funkcji.

```
>>> from sklearn.metrics import classification_report
>>> classification_report(y_test,  yknn_pred_test)
```

Poniżej przedstawimy podsumowanie trafności (oczywiście będziemy opierali się na próbie testowej) dla czterech omówionych klasyfikatorów[11]:

- KNN (3 sąsiadów),
- liniowa funkcja dyskryminacyjna,
- naiwny klasyfikator bayesowskiego,
- regresja logistyczna.

W tym celu wykorzystamy:

```
>>> for imo in [knn_3, lda, bayes, logit]:
...     print(imo)
...     print(classification_report(y_test, imo.predict(df_
test)))
...

KNeighborsClassifier(algorithm='auto', leaf_size=30, me-
tric='minkowski',
                     metric_params=None, n_jobs=None, n_ne-
ighbors=3, p=2,
                     weights='uniform')
              precision    recall  f1-score   support

           0       0.80      0.94      0.87        35
           1       0.98      0.92      0.95       104

    accuracy                           0.93       139
   macro avg       0.89      0.93      0.91       139
weighted avg       0.94      0.93      0.93       139

LinearDiscriminantAnalysis(n_components=None, priors=None,
shrinkage=None,
                           solver='svd', store_covarian-
ce=False, tol=0.0001)
              precision    recall  f1-score   support

           0       0.97      0.91      0.94        35
           1       0.97      0.99      0.98       104
```

[11] Zakładamy, że nazwy modeli pozostają takie jakich używalismy we wcześniejszych rozdziałach, tj. 'knn_3', 'lda', 'bayes' oraz 'logit'.

```
    accuracy                           0.97       139
   macro avg       0.97      0.95      0.96       139
weighted avg       0.97      0.97      0.97       139

GaussianNB(priors=None, var_smoothing=1e-09)
              precision    recall  f1-score   support

           0       0.92      0.94      0.93        35
           1       0.98      0.97      0.98       104

    accuracy                           0.96       139
   macro avg       0.95      0.96      0.95       139
weighted avg       0.96      0.96      0.96       139

LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_
iter=100,
                   multi_class='auto', n_jobs=None, penal-
ty='l2',
                   random_state=None, solver='lbfgs',
tol=0.0001, verbose=0,
                   warm_start=False)
              precision    recall  f1-score   support

           0       0.81      0.97      0.88        35
           1       0.99      0.92      0.96       104

    accuracy                           0.94       139
   macro avg       0.90      0.95      0.92       139
weighted avg       0.94      0.94      0.94       139
```

Spójrzmy na najprostszy wskaźnik, jakim jest trafność (*accuracy*). Biorąc pod uwagę ten wskaźnik najlepszą zdolność klasyfikacji ma model liniowej funkcji dyskryminacyjnej oraz klasyfikator Bayesa. Modele te dobrze klasyfikują obserwacje y=1 (w omawianym zbiorze danych oznacza to zmianę łagodną). Natomiast jeśli jesteśmy zainteresowani skutecznością wykrywania zmian złośliwych, wybierzemy model regresji logistycznej, który charakteryzuje się aż 97% czułością („recall'' dla wiersza y=1). Natomiast w 8% model ten generuje fałszywe alarmy, tzn. przewiduje, że zmiana będzie złośliwa podczas gdy w rzeczywistości jest łagodna („recall'' dla wiersza y=0 wynosi 0,92).

Z uwagi na cel książki, unikaliśmy szczegółowego omówienia własności teoretycznych. Poniżej przedstawimy podsumowanie, zaznaczając, że po części wynika ono z własności teoretycznych, a po części jest ona subiektywną kompilacją wyników uzyskanych w literaturze z własnymi doświadczeniami w stosowaniu tych klasyfikatorów. Dlatego porównanie to należy traktować ostrożnie, jako punkt wyjścia.

Tab. 5.1 Porównanie własności metod klasyfikacyjnych

Własność	KNN	LDA	Bayes	Logit
Nieliniowości	TAK	NIE	NIE	NIE
Skorelowane zmienne wejściowe	TAK	TAK	NIE	TAK
Obserwacje odstające	Słabo (zależnie od liczby najbliższych sąsiadów)	Dobrze	Słabo	Dobrze
Nadmierna liczba zmiennych wejściowych	Słabo	Średnio	Dobrze	Dobrze
Niedostateczna liczba zmiennych wejściowych	Średnio	Słabo	Średnio	Słabo

W przypadku gdy rzeczywiste zależności są silnie nieliniowe, najlepszą skuteczność ma klasyfikator najbliższych sąsiadów. W takim przypadku można też sprawdzić klasyfikator bayesowski oparty o zmienne poddane procedurze standaryzacji i rozkład inny niż normalny (np. t-Studenta albo log-normalny). Jeśli w próbie występują obserwacje odstające, dobrym rozwiązaniem może być użycie liniowej funkcji dyskryminacyjnej albo modelu regresji logistycznej. Jeśli mimo wszystko zdecydujemy się na użycie klasyfikatora KNN, użyjmy większej liczby „najbliższych sąsiadów" ($K > 10$).

Dane, którymi operujemy często charakteryzują się silnym skorelowaniem zmiennych objaśniających (tzw. przybliżona współliniowość). Jeśli taka korelacja jest umiarkowana, nie pogarsza to własności większości metod klasyfikacyjnych. W takim przypadku odradzalibyśmy tylko użycie klasyfikatora Bayesa, który w swojej konstrukcji zakłada niezależność zmiennych wejściowych.

Niedostateczna liczba zmiennych wejściowych (pominięcia) zawsze pogarsza jakość klasyfikacji. Relatywnie bardziej odporny na ten przypadek jest klasyfikator bayesowski. Warto pamiętać, że przypadek odwrotny — nadmierna liczba zmiennych wejściowych (użycie zmiennych, które nie są powiązane ze zmienną wejściową) również może powodować gorszą jakość klasyfikacji. Szczególnie narażone na skutki nadmiaru zmiennych wejściowych są klasyfikatory KNN i LDA, a z kolei używając prostego klasyfikatora bayesowskiego lub regresji logistycznej, minimalizujemy skutki złego doboru zmiennych wejściowych.

5.7 Klasyfikacja wieloklasowa

W rozdziałach 5.2–5.6 omawialiśmy przypadek najprostszy — klasyfikację binarną. Często zagadnienie jest nieco bardziej skomplikowane i obserwacja jest klasyfikowana do jednej z kilku grup (kategorii). Zupełnie tak jak w życiu, gdy odpowiedzi TAK/NIE nie wystarczają, możemy wskazać więcej możliwości (jed-

nak w przypadku klasyfikacji, liczba możliwości jest skończona i pojedyncza obserwacja może należeć wyłącznie do jednej kategorii).

W takim przypadku najprostszą możliwością jest redukcja odpowiedzi kilkuwariantowej do kilku pytań TAK/NIE. Przykładowo, jeśli pytamy klienta hotelu czy wybierze pokój (1) jednoosobowy, (2) dwuosobowy z osobnymi łóżkami, (3) dwuosobowy z jednym łóżkiem czy (4) duży apartament. Równie dobrze możemy zapytać (1) czy wybiera Pan/Pani pokój jednoosobowy, (2) czy wybiera Pan/Pani pokój dwuosobowy z osobnymi łóżkami, itd. W ten sposób odpowiedź na te cztery pytania[12] określą nam wybór klienta. Takie „rozbicie" modeli na ciąg modeli klasyfikacji binarnej gdzie modelujemy „wybraną kategorię kontra pozostałe" jest niekiedy stosowane (takie podejście w literaturze anglojęzycznej nazywane jest *one hot encoding*), choć konstrukcja osobnych modeli powoduje, że tracimy część informacji. Dlatego lepszym rozwiązaniem jest skorzystanie z modelu przewidzianego do klasyfikacji wieloklasowej.

Okazuje się, że większość omówionych metod klasyfikacji działa identycznie lub niemal identycznie w przypadku klasyfikacji wieloklasowej. Jest tak w przypadku metody *K* najbliższych sąsiadów (przy czym należy rozważyć liczbę *K* uniemożliwiającą „remis"), prostego klasyfikatora bayesowskiego oraz liniowej analizy dyskryminacyjnej. Natomiast regresja logistyczna w przypadku wieloklasowym wymaga zastosowania zupełnie innych algorytmów (lub, mówiąc językiem statystyki - metod estymacji). Najbardziej popularne metody to regresja logistyczna wielomianowa dla zmiennej uporządkowanej, gdy dla zmiennej wyjściowej możemy sensownie wyznaczyć wartości „większe" i „mniejsze" (np. wykształcenie, lot kategorią pierwszą/biznes/ekonomiczną, wartości na skali Likerta itp.), a w przeciwnym przypadku - zmiennej nieuporządkowanej (zob. Gruszczyński, 2012, rozdz. 4 i 5).

W przypadku miar jakości klasyfikacji możemy bezpośrednio użyć np. funkcji obliczającej macierz błędów. Bez przeszkód możemy też użyć .**classification_report()**.

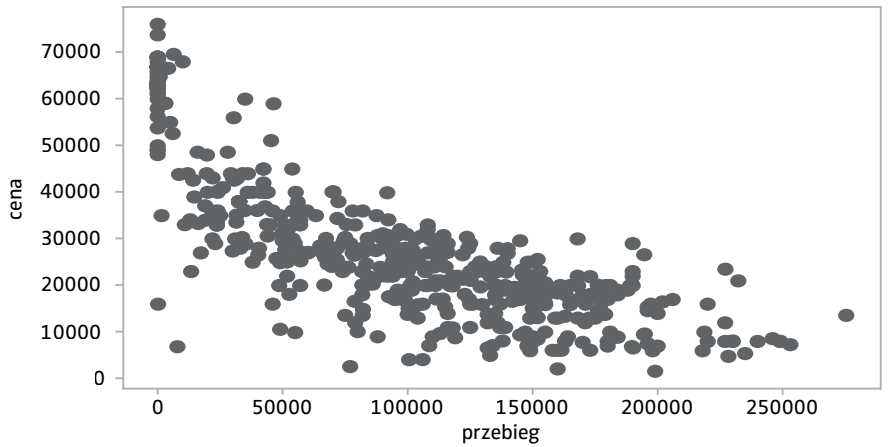
5.8 Model regresji

Podobnie jak w przypadku klasyfikacji, w regresji także mamy **zmienną objaśnianą** (wyjściową, y), której poziom będziemy uzależniać od **zmiennej/zmiennych objaśniających** (predyktorów, zmiennych wejściowych — X). Przypomnijmy też, że w problemie regresji zmienna objaśniana jest zmienną ciągłą, tzn. predyktory pozwalają przewidzieć (określić) jej poziom. Prostym przykładem wykorzystania regresji może być analiza popytu na jakieś dobro i jego ceny. W tym przypadku popyt jest zmienną objaśnianą, której poziom zależy od ceny tego dobra.

[12] W zasadzie do oznaczenia wyboru z czterech kategorii wystarczą odpowiedzi na trzy pytania — jeśli klient nie wybierze odpowiedzi (1)–(3), oznacza to że preferuje wariant (4).

Podobnie jak w przypadku klasyfikacji, zmienne X mogą być ciągłe lub dyskretne, choć w przykładach będziemy rozważali jedynie ciągłe zmienne wejściowe. Przedmiotem naszej analizy w tym rozdziale będą dane zgromadzone z serwisu *gratka.pl* za pomocą metod maszynowego pobierania danych ze stron internetowych omówionych w rozdziale 3. Dane te zostały wstępnie przetworzone (oczyszczone) (por. Załącznik do rozdz. 3). Na Rys. 5.8 przedstawiono ceny samochodów zamieszczone w ogłoszeniach oraz ich przebieg. Na jego podstawie możemy zaobserwować, że zależność między przebiegiem samochodu a jego ceną nie jest deterministyczna, co znaczy, że punkty nie układają się w funkcję, np. linię prostą, której każdy punkt moglibyśmy opisać równaniem. Dlatego też znając poziom zmiennej X możemy jedynie określić przybliżony poziom zmiennej y. Jednocześnie na Rys. 5.8 możemy dostrzec, że wraz ze wzrostem przebiegu, cena samochodu maleje. Możemy też wskazać jakiego poziomu ceny oczekiwać patrząc na grupę samochodów o tym samym przebiegu, tzn. możemy zaobserwować rozkład wartości ceny dla samochodów o konkretnym, danym, przebiegu. Podsumowując, co prawda nie możemy za pomocą regresji wskazać konkretnych wartości zmiennej objaśnianej, możemy jednak powiedzieć jakich wartości należy oczekiwać, innymi słowy jaka będzie średnia wartość ceny dla samochodów z danym przebiegiem.

Rys. 5.8 Wykres rozrzutu ceny samochodu i jego przebiegu



Liniowy model regresji informuje nas jak zmienia się oczekiwana (średnia) wartość zmiennej objaśnianej w zależności od zmiennej X (jako funkcja zmiennej X).

$$y_i = f(X_i) + \varepsilon_i,$$

gdzie ε to składnik losowy.

Model regresji liniowej zakłada, że związek łączący zmienną objaśniającą i objaśnianą jest prostoliniowy, tzn. zakładamy, że funkcja $f(X)$ jest funkcją liniową, co możemy dalej zapisać:

$$y_i = \beta_0 + \beta_1 X_i + \varepsilon_i.$$

Wyrażenie $\beta_0+\beta_1 X_i$ określa wartość oczekiwaną, którą zapisujemy jako $E(Y|X)$ i jest to część zmienności, która jest opisywana przez model. Różnica pomiędzy y_i a $\beta_0+\beta_1 X_i$ to „zakłócenie'', tj. błąd szacunku, który przedstawia tę część zmienności Y, która nie jest opisywana przez ujęte w modelu predyktory. Błąd ten może wynikać z różnych przyczyn, m. in. z faktu, iż zwykle nie mamy danych o wszystkich potencjalnych predyktorach opisywanego zjawiska, z błędów pomiaru czy np. z cech specyficznych dla konkretnej obserwacji. ε_i pokazuje w jaki sposób faktycznie obserwowana wartość Y_i odchyla się od swojej wartości oczekiwanej (średniej).

Równoważnie model przedstawiony powyżej możemy zapisać jako:

$$E(Y \mid X = x) = \beta_0 + \beta_1 x,$$

w szczególności dla X=0

$$E(Y \mid X = 0) = \beta_0 + \beta_1 \cdot 0 = \beta_0.$$

β_0 stanowi więc wartość oczekiwaną Y jeśli X przyjmuje wartość zero.

$$E(Y \mid X = 1) = \beta_0 + \beta_1 \cdot 1$$

$$E(Y \mid X = 2) = \beta_0 + \beta_1 \cdot 2$$

Zauważmy teraz, że jeśli X zwiększa się o jednostkę, wartości zmiennej Y zmieniają się o β_1. Parametr ten jest nachyleniem funkcji $f(X)$. Parametr β_0 jest natomiast wyrazem wolnym.

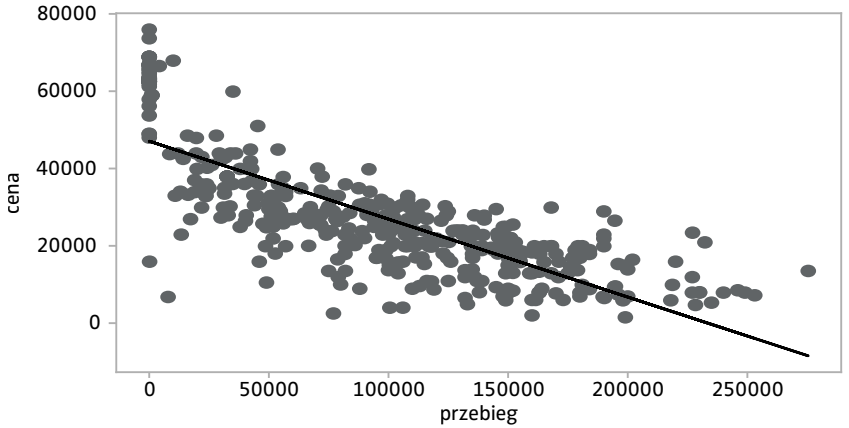
Przekładając „język'' modelu na dane empiryczne, naszym celem będzie zbudowanie modelu objaśniającego kształtowanie się ceny samochodu w zależności od jego przebiegu, a zatem oszacowanie parametrów β_0 i β_1. Jednak do tego celu, jak to już zostało wyjaśnione w rozdziale 5.1.1, nie wykorzystamy całego zbioru danych, tylko podzielimy dostępne dane na próby uczącą i testową. Następnie do wyznaczenia parametrów modelu użyjemy danych z próby uczącej tak, aby móc dokonać weryfikacji jego zdolności predykcyjnych

wykorzystując dane z próby testowej. Dane wykorzystane w przykładach przechowujemy w ramce danych `auto`.

```
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.model_selection import train_test_split
>>> y = auto['cena']
>>> x = auto[['przebieg']]
>>> x_tren, x_test, y_tren, y_test = train_test_split(x, y,
test_size=0.2, random_state=0)
>>> model = LinearRegression()
>>> model.fit(x_tren, y_tren)
>>> print('wyraz wolny:', model.intercept_)
>>> print('nachylenie:', model.coef_)
```

Metoda **.fit()** służy do dopasowania modelu do danych. Wykorzystany model dokonuje oszacowań parametrów za pomocą klasycznej metody najmniejszych kwadratów (KMNK, *Ordinary Least Squares* — OLS), która minimalizuje sumę kwadratów odchyleń wartości obserwowanych w zbiorze uczącym od wartości prognozowanych przez model. Wyznaczone parametry β_0 i β_1 są zapisane w polach **.intercept_** oraz **.coef_**. Otrzymane wartości to w przybliżeniu 47068 (wyraz wolny) oraz -0,2 (parametr odpowiadający za nachylenie). Na Rys. 5.9 dokonujemy graficznej prezentacji oszacowanego modelu liniowego, aby dokonać przybliżonej (wzrokowej) oceny poprawności jego dopasowania do danych empirycznych.

Rys. 5.9 Oszacowany model liniowy — dopasowanie do danych obserwowanych (próba ucząca)



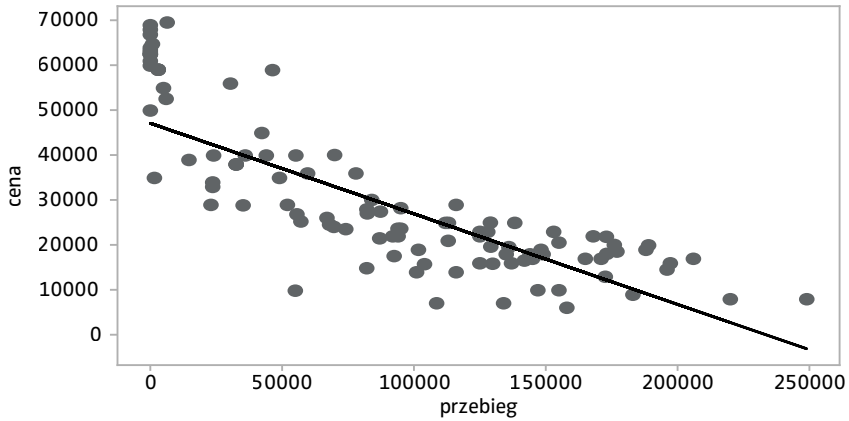
Na podstawie Rys. 5.9 możemy stwierdzić, że oszacowany model (czarna linia) dość dobrze dopasował się do chmury punktów (obserwacji w próbie uczącej). Jak wiemy, w uczeniu nadzorowanym, bardziej jednak będą nas in-

teresowały zdolności predykcyjne tego modelu poza próbą uczącą (por. rozdz. 5.1.1).

Na Rys. 5.10 dokonujemy graficznej prezentacji obserwacji z próby testowej (szare punkty) oraz oczekiwanej (prognozowanej) na podstawie naszego modelu ceny samochodu (czarna linia[13]).

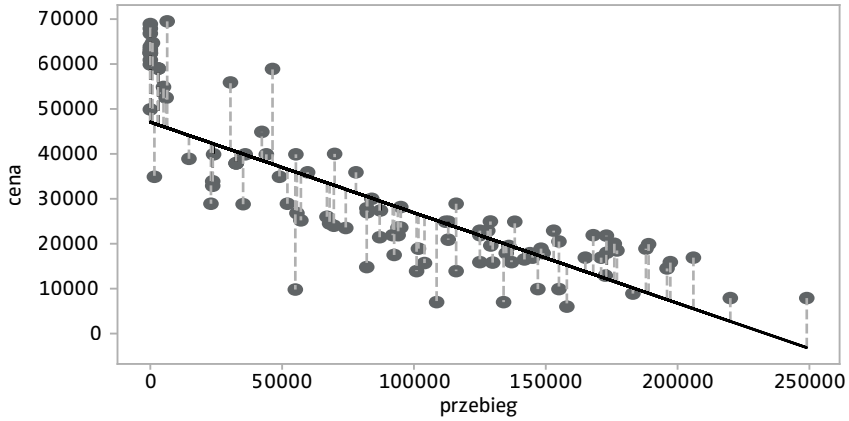
Oprócz oceny przybliżonej (na podstawie Rys. 5.10), chcielibyśmy się posługiwać bardziej precyzyjnymi miarami stopnia dopasowania prognoz do wartości faktycznie obserwowanych. Jak się za chwilę okaże, będą one zależeć od błędów szacunku $(y_i - \hat{y}_i)$, czyli różnic między wartościami obserwowanymi a wartościami wypredykowanymi przez model. Różnice te przedstawiliśmy na Rys. 5.11 za pomocą przerywanych linii.

Rys. 5.10 Oszacowany model liniowy — dopasowanie do danych obserwowanych (próba testowa)



[13] Punkty układają się w prostą, ponieważ dokonaliśmy predykcji ceny samochodu dla każdej wartości przebiegu z podanego zakresu.

Rys. 5.11 Oszacowany model liniowy — błędy szacunków (próba testowa)



Pierwszą z miar dopasowania modelu do danych jest współczynnik determinacji nazywany również współczynnikiem R^2 („R kwadrat"). Określa on jaka część zmienności zmiennej zależnej (w naszym przypadku ceny samochodu) jest objaśniana przez model. Współczynnik ten obliczamy za pomocą wzoru:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y}_i)^2},$$

gdzie symbolem $\overline{y}_i$ oznaczyliśmy średnią wartość zmiennej zależnej (w analizowanym zbiorze danych, w tym przypadku — próbie testowej). Współczynnik R^2 przyjmuje wartości z przedziału <0;1>. Im wyższa jego wartość, tym lepiej model jest dopasowany do danych.

W celu jego obliczenia korzystamy z metody .**score()**, której argumentami są odpowiednio wartości zmiennej objaśniającej i objaśnianej:

```
>>> r_sq = model.score(x_test, y_test)
>>> print('wsp. dopasowania:', r_sq)
```

Okazuje się, że dla próby testowej wynosi on w przybliżeniu 0,68, co można uznać za wynik satysfakcjonujący w przypadku modelu z jedną zmienną objaśniającą. Oczywiście, moglibyśmy również chcieć ocenić jakie było dopasowanie modelu do danych z próby uczącej:

```
>>> r_sq_tren = model.score(x_tren, y_tren)
>>> print('wsp. dopasowania:', r_sq_tren)
```

W naszym przypadku otrzymaliśmy wynik 0,63, t.j. dopasowanie zbliżone do tego w próbie testowej, dlatego możemy wysnuć wniosek o braku problemu „przeuczenia", który objawiałby się nadmiernym (bliskim jedności) współczynnikiem R^2 dla próby uczącej i zdecydowanie niższym w przypadku próby testowej. Zjawisko to szerzej omawiamy w rozdziale 5.1.1.

Innymi miarami „jakości" modelu są: MAE (*Mean Squared Error* — średni błąd absolutny), określony wzorem:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i\right|$$

oraz RMSE (*Root Mean Squared Error* — pierwiastek ze średniego błędu kwadratowego):

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2}.$$

Są to miary błędów, stąd też pożądane są ich niskie wartości. Możemy je obliczyć w następujący sposób:

```
>>> from sklearn import metrics
>>> from math import sqrt
>>> y_pred_test = model.predict(x_test)
>>> print('MAE:', metrics.mean_absolute_error(y_test,y_
pred_test))
>>> print('MSE:', metrics.mean_squared_error(y_test, y_
pred_test))
>>> print('RMSE:', sqrt(metrics.mean_squared_error(y_test,
y_pred_test)))
MAE: 7838.33336522
MSE: 96516410.4582
RMSE: 9824.27658702
```

Oczywiście, model uwzględniający tylko jedną zmienną objaśnianą (predyktora) jest modelem dość prostym. Zwykle wykorzystywane są modele, w których liczba predyktorów jest zdecydowanie większa, a poszczególne zmienne są dobrane w ten sposób, aby przekazywały jak najwięcej informacji na temat analizowanego zjawiska. Do naszego modelu uzależniającego cenę samochodu od jego przebiegu wprowadzimy więc drugi predyktor, a mianowicie informację o roku produkcji. Dlatego też definiujemy na nowo zbiór predyktorów. W kolejnych krokach ponownie dzielimy dane na zbiory uczący i testowy, po czym dokonujemy dopasowania modelu do danych z próby uczącej:

```
>>> y = auto['cena']
>>> x = auto[['przebieg', 'rok']]
```

```
>>> x_tren, x_test, y_tren, y_test = train_test_split(x, y,
test_size=0.2, random_state=0)
>>> model = LinearRegression()
>>> model.fit(x_tren, y_tren)
>>> print('wyraz wolny:', model.intercept_)
wyraz wolny: -3751558.3329
>>> print('parametry:', model.coef_)
parametry: [ -1.06793405e-01   1.88266770e+03]
```

Zwracamy uwagę, że tym razem oprócz wyrazu wolnego, otrzymaliśmy oszacowania dwóch parametrów. Odpowiadają one danym z kolejnych kolumn zbioru X, czyli zmiennej *przebieg* i zmiennej *rok*. Widzimy również, że otrzymane znaki oszacowań są intuicyjne i wskazują na ujemną (dodatnią) zależność pomiędzy przebiegiem (rokiem produkcji) samochodu a jego ceną. Wyznaczmy ponownie miary dopasowania (nowego) modelu do danych testowych. Wynoszą one (po zaokrągleniu):

- R^2: 0.80;
- MAE: 6043.8;
- MSE: 61274916.8;
- RMSE: 7827.8.

W porównaniu do modelu z jedną zmienną objaśniającą widzimy zatem poprawę „jakości" dopasowania modelu do danych wyrażającą się we wzroście współczynnika R^2, przy spadku wartości błędów.

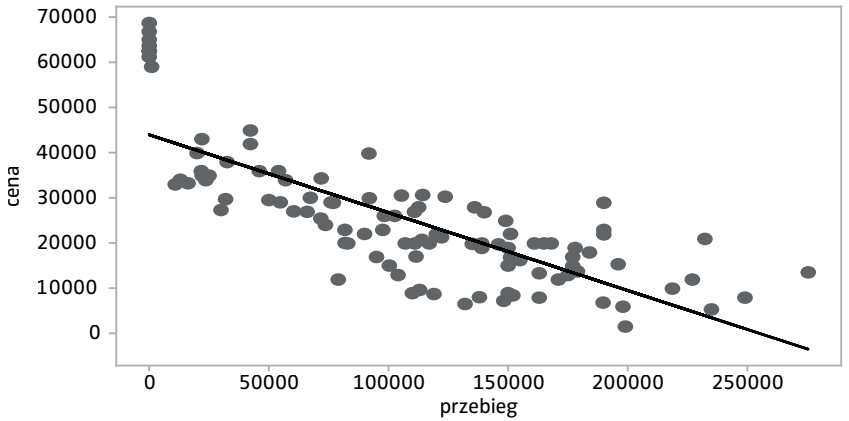
5.9 Uczenie maszynowe — dalsze zagadnienia

Dotychczas przedstawiliśmy podstawowe metody klasyfikacji i regresji. Mimo, że nie sposób przedstawić wszystkich metod, prezentujemy również nieco bardziej zaawansowane zagadnienia uczenia maszynowego. Materiał zawarty w tym rozdziale może służyć pogłębieniu wiedzy, szczególnie Czytelników o lepszym przygotowaniu z zakresu matematyki i statystyki.

Pierwszym zagadnieniem są obserwacje nietypowe (odstające, *outliers*). Jak już wspomniano w Załączniku do rozdziału 3, często już na początkowym etapie analizy możliwe są do wychwycenia pewne obserwacje, które z różnych przyczyn „nie pasują" do pozostałych. Jako przykład wskazaliśmy wówczas wartość -99 dla przebiegu lub roku produkcji samochodu, ponieważ oczekujemy w przypadku przebiegu liczb nieujemnych, a jeśli chodzi o rok produkcji — wartości z określonego przedziału. Nieco innym przypadkiem byłaby np. obserwacja, w której wartość przebiegu jest znacząco wyższa (choć nadal prawdopodobna) niż pozostałe obserwacje w próbie uczącej i jednocześnie sa-

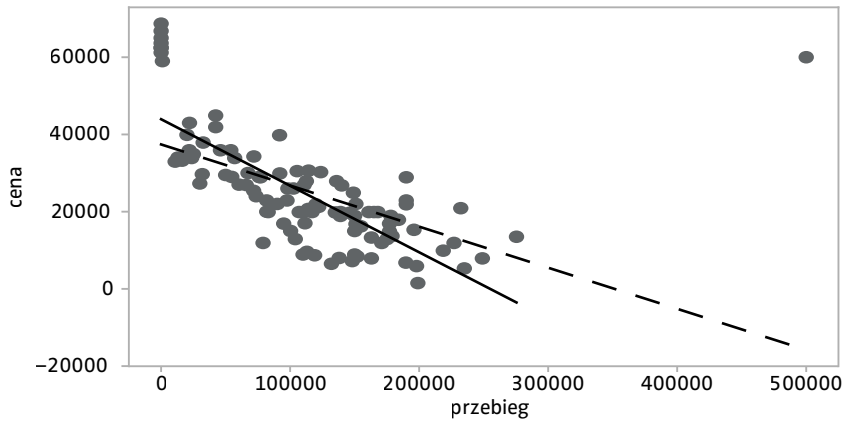
mochód ma dość wysoką cenę. Prześledźmy ten przypadek dla części danych wykorzystanych uprzednio w przykładzie regresji liniowej[14].

Rys. 5.12 Dopasowanie do danych obserwowanych (próba ucząca)



Na Rys. 5.12 czarnym kolorem przedstawiono dopasowaną linię regresji. Następnie, próbę uczącą rozszerzono o obserwację o wysokim przebiegu oraz relatywnie wysokiej cenie. Można ją zaobserwować w prawym górnym rogu na Rys. 5.13.

Rys. 5.13 Dopasowanie do danych obserwowanych (próba ucząca z obserwacją nietypową)



[14] Podział na próbę uczącą i testową dla omawianego przykładu można uzyskać poleceniem: `x_tren, x_test, y_tren, y_test = train_test_split(x, y, test_size=0.8, random_state=0)`.

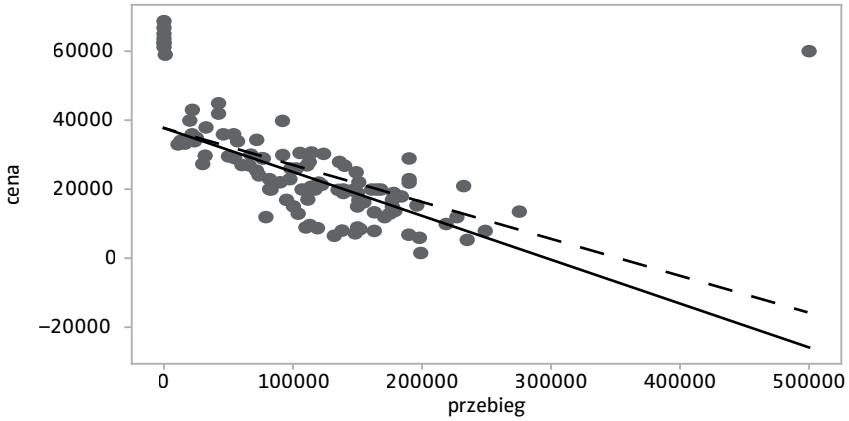
Na Rys. 5.13 linia przerywana to linia regresji modelu dopasowanego do zbioru uczącego rozszerzonego o obserwację nietypową, zaś ciągła to linia regresji dopasowana do zbioru bez tejże obserwacji. Okazuje się, że wprowadzenie tej jednej obserwacji znacząco wpłynęło na nachylenie linii regresji, które obecnie jest bardziej łagodne. Obserwację taką nazywa się wpływową. Konsekwencją zmienionych oszacowań parametrów modelu po włączeniu obserwacji nietypowej są odpowiednio wyższe błędy w próbie testowej i tym samym niższa zdolność prognostyczna modelu. Istnieje kilka rozwiązań tego problemu. Jedno z nich polega na usunięciu obserwacji nietypowych z próby. Do zalet tego rozwiązania niewątpliwie należy prostota, natomiast do wad — trudności w identyfikacji obserwacji nietypowych oraz utrata informacji. Innym rozwiązaniem jest wybór metody odpornej na obserwacje nietypowe. Metody te pozwalają zachować obserwacje nietypowe w zbiorze uczącym, jednocześnie ograniczając ich wpływ na oszacowania parametrów. Jedną z nich jest metoda najmniejszych odchyleń bezwzględnych (LAD — *Least Absolute Deviation*), która minimalizuje sumę wartości bezwzględnych błędów, podczas gdy metoda regresji liniowej zaprezentowana w rozdziale 5.8 minimalizuje sumę kwadratów błędów (zob. np. Greene, 2000, s. 448–449). LAD jest szczególnym przypadkiem regresji kwantylowej (*Quantile regression*) — dla kwantyla rzędu 0,5 (mediany).

W przykładzie poniżej zastosujemy LAD do oszacowania zależności pomiędzy przebiegiem a ceną samochodu wykorzystując zbiór uczący rozszerzony o obserwację nietypową. Posłużymy się biblioteką `statsmodels` i dokonamy oszacowań modelu zawierającego stałą.

```
from statsmodels.regression import quantile_regression
from statsmodels.tools import add_constant
x = add_constant(x)
lad = quantile_regression.QuantReg(y, x)
print(lad.fit().params)
const       37710.276161
przebieg       -0.127168
dtype: float64
```

Na Rys. 5.14 linią ciągłą przedstawiono linię regresji modelu dopasowanego do danych. Przerywana linia oznacza linię regresji wyznaczoną z wykorzystaniem klasycznej metody najmniejszych kwadratów. Jak wynika z Rys. 5.14, nachylenie linii odpowiadającej metodzie najmniejszych odchyleń (LAD) jest większe (linia jest bardziej stroma) niż w przypadku metody KMNK. Oznacza to ograniczenie wpływu obserwacji nietypowej na oszacowania parametrów modelu.

Rys. 5.14 Linie regresji oszacowane metodami LAD oraz KMNK (próba ucząca z obserwacją nietypową)



Drugim ważnym i coraz bardziej popularnym zagadnieniem są metody tzw. regularyzacji. Regularyzacja jest dodatkowym ograniczeniem narzucanym na metody uczenia nadzorowanego. Ogólna koncepcja regularyzacji zakłada zmniejszenie siły wpływu zmiennych objaśniających, w porównaniu z metodami standardowymi (np. klasyczną metodą najmniejszych kwadratów). Dzięki temu, mając dane zawierające dużą ilość „szumu'' (losowych zakłóceń), regularyzacja sprawia, że modele uczenia maszynowego w mniejszym stopniu reagują na „szum'', dzięki czemu parametry nie są drastycznie przeszacowane albo niedoszacowane (w statystyce mówimy, że następuje redukcja wariancji kosztem niewielkiego obciążenia estymacji). Kosztem tego zabiegu jest mniejsza dokładność dopasowania modelu do danych wyjściowych w próbie treningowej. W najprostszym ujęciu stopniowo redukujemy listę zmiennych objaśniających, pozostawiając jedynie te, które silnie wpływają na zmienną objaśnianą[15]. Nowsze metody (wymienimy jedynie nazwy: LASSO, LARS, Elastic-Net) już na etapie estymacji parametrów kierują nie tylko dopasowaniem linii regresji (jak to ma miejsce w przypadku metody najmniejszych kwadratów), ale dodatkowo biorą pod uwagę łączną siłę wpływu zmiennych objaśniających. Przykładowo, w metodzie LASSO ograniczenie powoduje, że zeruje się wybrane parametry — czyli podobnie jak w przypadku regresji krokowej, metoda ta pozwala pozostawić najważniejsze zmienne (przegląd tej i innych metod regu-

[15] Choć w książce nie opisujemy testowania istotności statystycznej parametrów, należy dodać, że zwykle silny wpływ oznacza to samo co „parametr istotny statystycznie''. Taka redukcja zmiennych nieistotnych statystycznie stosowana jest w ekonometrii i statystyce od dawna (np. modelowanie „od ogółu do szczegółu'', ang. *general-to-specific modelling*, zob. Welfe, 2018). W uczeniu maszynowym bardzo zbliżoną procedurę określa się zwykle jako „regresja krokowa wsteczna'', ang. *backward stepwise regression*.

laryzacji w: James i in., 2017, rozdz. 6.2). Regularyzację przedstawiliśmy dla problemów regresji, ale jej idea jest stosowana także w zagadnieniach klasyfikacji (np. Kwiatkowski, 2019).

Zarówno ograniczenie wpływu poprzez usunięcie zmiennych wejściowych, jak i zastosowanie specjalnych metod regresji z regularyzacją, pogarsza dopasowanie modelu do danych w próbie uczącej. Jak pamiętamy (rozdz. 5.1.1), dobrze zbudowany model zapewnia niski błąd predykcji dla obserwacji spoza próby uczącej. Okazuje się, że mimo pogorszenia błędów w próbie uczącej, metody regularyzacji pozwalają poprawić błędy predykcji dla próby testowej. Ta cecha spowodowała, że metody regularyzacji weszły już do kanonu technik uczenia maszynowego.

Literatura

Do rozdz. 1

Aczel A. (2017). *Statystyka w zarządzaniu*, Wydawnictwo Naukowe PWN, Warszawa.

Bengio Y., Courville A., Goodfellow I. (2018). *Deep learning. Systemy uczące się*, Wydawnictwo Naukowe PWN, Warszawa.

Boschetti A., Massaron L. (2017). *Python. Podstawy nauki o danych*, Helion, Gliwice.

Gabrusewicz W. (2019). *Metody analizy finansowej przedsiębiorstwa*, Polskie Wydawnictwo Ekonomiczne, Warszawa.

Gorelick M., Ozsvald I. (2020). *High Performance Python: Practical Performant Programming for Humans*, O'Reilly Media, Beijing etc.

Meyer C. (2000). *Matrix Analysis and Applied Linear Algebra*, SIAM, Philadelphia.

Provost F., Fawcett T. (2019). *Analiza danych w biznesie*, Helion, Gliwice.

Slatkin S. (2015). *Efektywny Python. 59 sposobów na lepszy kod*, Helion, Gliwice.

Do rozdz. 2

Biecek P. (2014). *Przewodnik po pakiecie R*, Oficyna Wydawnicza GIS, Wrocław.

Jaworski M., Ziadé T. (2017). *Profesjonalne programowanie w Pythonie. Poziom ekspert*, Helion, Warszawa.

Kopczewska K., Kopczewski T., Wójcik P., Marcinkowska-Lewandowska W. (2016). *Metody ilościowe w R. Aplikacje ekonomiczne i finansowe*, CeDeWu, Warszawa.

Lutz M. (2011). *Python. Wprowadzenie*, Helion, Gliwice.

McIlroy P. (1993). *Optimistic Sorting and Information Theoretic Complexity*, [w:] Proceedings of the Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Austin, Texas, USA, s. 467–474.

Peters T. (2002). *Timsort — Python*, https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

Zhang H., Meng B., Liang Y. (2016). *Sort race*, https://arxiv.org/ftp/arxiv/papers/1609/1609.04471.pdf.

Do rozdz. 3

Baranowski P., Komor M., Wójcik S. (2018). Whose feedback matters? Empirical evidence from online auctions, *Applied Economics Letters*, 25(17), s. 1226–1229.

Beręsewicz M. (2016). *Internet data sources for real estate market statistics*, Poznan University of Economics and Business, http://www.wbc.poznan.pl/dlibra/doccontent.

Brener A. (2019). Payment Service Directive II and Its Implications, [w:] *Disrupting Finance*, Palgrave Pivot, Cham, s. 103–119.

Cavallo A., Rigobon R. (2016). The billion prices project: Using online prices for measurement and research, *Journal of Economic Perspectives*, 30(2), s. 151–78.

Duckett J. (2015). *Javascript i Jquery. Interaktywne strony WWW dla każdego*, Helion, Gliwice.

Dusi S., Mercorio F., Mezzanzanica M. (2015). *Big data meets web job vacancies: trends, challenges and development directions*, [w:] Larsen C., Rand S., Schmid A., Mezzanzanica M., Dusi S. *Big Data and the Complexity of Labour Market Policies: New Approaches in Regional and Local Labour Market Monitoring for Reducing Skills Mismatches*, Rainer Hampp Verlag, s. 31–44.

Edelman B. (2012). Using internet data for economic research, *Journal of Economic Perspectives*, 26(2), s. 189–206.

Gierszewski T., Molisz W. (2014). Ataki DDoS — przegląd zagrożeń i środków zaradczych, *Przegląd Telekomunikacyjny + Wiadomości Telekomunikacyjne*, 8–9, s. 720–728.

Gundecha U. (2015). *Selenium i testowanie aplikacji. Receptury*, Helion, Gliwice.

Jarmin R. S. (2019). Evolving measurement for an evolving economy: Thoughts on 21st Century US economic statistics, *Journal of Economic Perspectives*, 33(1), s. 165–84.

Lucking-Reiley D., Bryan D., Prasad N., Reeves D. (2007). Pennies from eBay: The determinants of price in online auctions, *The Journal of Industrial Economics*, 55(2), s. 23–233.

Macias P., Stelmasiak D. (2019). *Food inflation nowcasting with web scraped data*, Narodowy Bank Polski, https://www.nbp.pl/publikacje/materialy_i_studia/302_en.pdf.

Maślankowski (2019). Pozyskiwanie i analiza danych na temat ofert pracy z wykorzystaniem big data, *Wiadomości Statystyczne*, 64(9), s. 60–74.

Mitchell R. (2015). *Web scraping with Python*, O'Reilly, Beijing etc.

Paduszyńska M., Pawlak B. (2020). Rynek usług płatniczych w Polsce w świetle zmian prawnych implementujących postanowienia dyrektywy PSD2, *Studia Prawno-Ekonomiczne*, (114), s. 333–349.

Rother K. (2018). *Python dla profesjonalistów. Debugowanie, testowanie i utrzymywanie kodu*, Helion, Gliwice.

Schafer S. (2013). *HTML, XHTML i CSS. Biblia*, Helion, Gliwice.

Verma H., Verma G. (2019). Prediction Model for Bollywood Movie Success: A Comparative Analysis of Performance of Supervised Machine Learning Algorithms, *The Review of Socionetwork Strategies*, 14(1), s. 1–17.

Do rozdz. 4

Beckwith R., Miller G.A., Tengi R. (1993). *Design and implementation of the WordNet lexical database and searching software*, http://wordnetcode.princeton.edu/5papers.pdf.

Bird S., Klein E., Loper E. (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*, O'Reilly Media, Sebastopol, CA.

Cieślewicz J., Pelikant A. (2009). Reprezentacja i wyszukiwanie dokumentów tekstowych w bazach danych, *Studia Informatica*, 30(2A), s. 259–271.

Fellbaum C. (1990). *English verbs as a semantic net*, http://wordnetcode.princeton.edu/5papers.pdf.

Gładysz A. (2016). Przegląd zastosowań analizy text miningowej, *Autobusy: technika, eksploatacja, systemy transportowe*, 17(12), s. 1742–1746.

Gross D., Miller K.J. (1993/1990). *Adjectives in wordnet*, http://wordnetcode.princeton.edu/5papers.pdf.

Hearty J. (2016). *Advanced Machine Learning with Python*, Packt Publishing Ltd., Birmingham.

Jurafsky D., Martin J.H., (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*, Wydanie 2, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Manning C.D., Raghavan P., Schütze H. (2009). *Introduction to information retrieval*, https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf.

Maziarz M., Piasecki M., Rudnicka E. (2014). Słowosieć: polski wordnet. Proces tworzenia tezaurusa, *POLONICA*, (34), s. 79–98.

Miller G.A. (1993/1990). *Nouns in WordNet: a lexical inheritance system,* http://wordnetcode.princeton.edu/5papers.pdf.

Miller G.A., Beckwith R., Fellbaum C., Gross D., Miller K.J. (1993/1990). *Introduction to WordNet: An on-line lexical database*, http://wordnetcode.princeton.edu/5papers.pdf.

Mykowiecka A. (2007). *Inżynieria lingwistyczna. Komputerowe przetwarzanie tekstów w języku naturalnym*, Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, Warszawa.

Porter M.F. (1980). *An algorithm for suffix stripping*, Program 14.3, s. 130–137.

Sarkar D. (2016). *Text Analytics with Python: A Practical Real-World Approach to Gaining Actionable Insights from Your Data*, Apress, Bangalore.

Do rozdz. 5

Bengio Y., Courville A., Goodfellow I. (2018). *Deep learning. Systemy uczące się*, Wydawnictwo Naukowe PWN, Warszawa.

Buczak A.L., Guven E. (2015). A survey of data mining and machine learning methods for cyber security intrusion detection, *IEEE Communications surveys & tutorials*, 18(2), s. 1153–1176.

Chinnamgari S. K. (2019). *R Machine Learning Projects: Implement supervised, unsupervised, and reinforcement learning techniques using R 3.5*, Packt Publishing Ltd., Birmingham, UK.

Crawford M., Khoshgoftaar T.M., Prusa J.D., Richter A.N., Al Najada H. (2015). Survey of review spam detection using machine learning techniques, *Journal of Big Data*, 2(1), s. 2–23.

Finlay S. (2010). *Credit Scoring, Response Modelling and Insurance Rating. A practical guide to forecasting consumer behavior*, Palgrave Macmillan, London, UK

Greene W. H. (2000). *Econometric analysis*, Prentice Hall, Upper Saddle River, N.J.

Gruszczyński M. (2012). *Mikroekonometria. Modele i metody analizy danych indywidualnych*, Oficyna Wolters Kluwer business, Warszawa.

James G., Witten D., Hastie T., Tibshirani R. (2017). *An introduction to Statistical Learning with Applications in R*, Springer, New York.

Kwiatkowski W. (2019). The regularization method in the classification task according to given examples, *Teleinformatics Review*, 7(25), 3–4, s. 3-13.

Li X., Kurita T. (2015). Nonlinear discriminant analysis using K nearest neighbor estimation, [w:] *2015 21st Korea–Japan Joint Workshop on Frontiers of Computer Vision (FCV)*, IEEE, s. 1–6.

Maddala G. S. (2006). *Ekonometria*, Wydawnictwo Naukowe PWN.

Mangasarian O.L., Street O.L., Wolberg W.H., (1995). Breast cancer diagnosis and prognosis via linear programming, *Operations Research*, 43(4), s. 570–577.

Raschka S. (2015). *Python machine learning*, Packt Publishing Ltd., Birmingham, UK.

Shmueli G. (2010). To explain or to predict?, *Statistical science*, 25(3), s. 289–310.

Street W.N., Wolberg W.H., Mangasarian O.L. (1993). Nuclear feature extraction for breast tumor diagnosis, *IS&T/SPIE 1993 International Symposium on Electronic Imaging: Science and Technology*, 1905, San Jose, CA.

Train K. (2009). *Discrete choice methods with simulation*, Cambridge University Press, Cambridge, etc.

Verbeke W., Martens D., Mues C., Baesens B. (2011). Building comprehensible customer churn prediction models with advanced rule induction techniques, *Expert Systems with Applications*, 38(3), s. 2354–2364.

Welfe A. (2018). *Ekonometria*, Polskie Wydawnictwo Ekonomiczne, Warszawa.

Spis rysunków

Spis tabel

Dr hab. Paweł Baranowski specjalizuje się w polityce pieniężnej, ekonometrii stosowanej oraz uczeniu maszynowym. Pracuje na stanowisku profesora nadzwyczajnego w Katedrze Ekonometrii Uniwersytetu Łódzkiego, a także w Instytucie Ekspertyz Ekonomicznych i Finansowych w Łodzi. W latach 2009–2018 pracował jako analityk w Narodowym Banku Polskim oraz Commerzbank A.G. Uczestniczył w szkoleniach i seminariach m.in. Międzynarodowego Funduszu Walutowego oraz Banku Anglii. Opublikował ponad 40 prac naukowych w kraju i zagranicą. Kierował lub brał udział w ogólnopolskich projektach badawczych, był też głównym wykonawcą modeli uczenia maszynowego wdrożonych w przedsiębiorstwach handlowych i usługowych. Za pracę doktorską otrzymał nagrodę im. prof. Witolda Kuli, a za całokształt osiągnięć naukowych i organizacyjnych odznakę „Za Zasługi dla Miasta Łodzi".

Dr Wirginia Doryń jest absolwentką Wydziału Ekonomiczno-Socjologicznego oraz Wydziału Matematyki i Informatyki Uniwersytetu Łódzkiego. Pracuje na stanowisku adiunkta w Katedrze Funkcjonowania Gospodarki Uniwersytetu Łódzkiego. Jej zainteresowania badawcze obejmują handel zagraniczny oraz zagadnienia leżące na styku ekonomii i ekologii. Jest autorką wielu publikacji naukowych. Uczestniczyła w licznych szkoleniach specjalistycznych, w tym kursach zaawansowanej analizy danych. Jest wykonawcą projektów badawczych wykorzystujących duże zbiory danych i metody przetwarzania tekstu. Za pracę doktorską otrzymała Nagrodę JM Rektora Uniwersytetu Łódzkiego.

ISBN 978-83-65605-16-0

9 788365 605160

IBG
INSTYTUT BADAŃ GOSPODARCZYCH