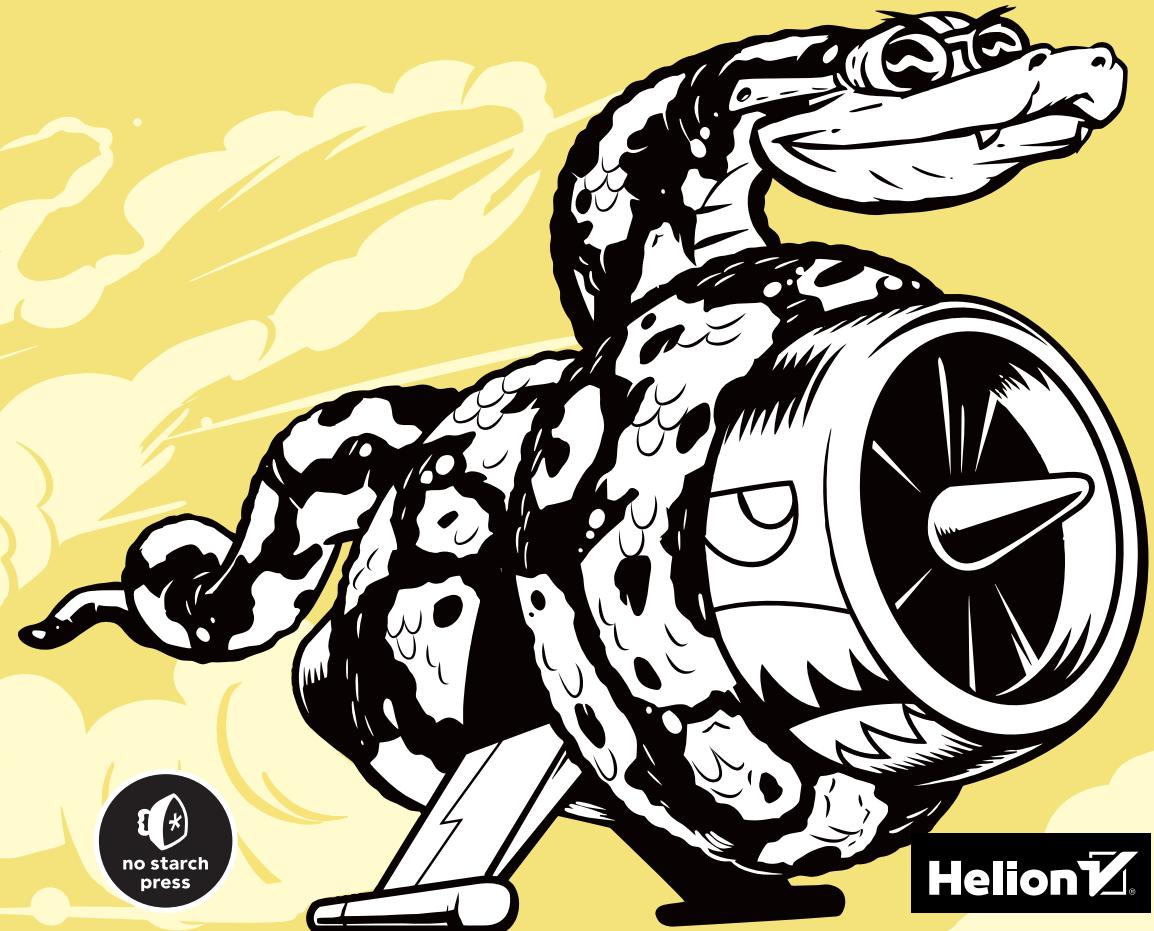


Wydanie III

# PYTHON

INSTRUKCJE  
DLA PROGRAMISTY

ERIC MATTHES



Helion 

Tytuł oryginału: Python Crash Course: A Hands-On, Project-Based Introduction to Programming, 3<sup>rd</sup> Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-0431-6

Copyright © 2023 by Eric Matthes. Title of English-language original: *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, 3<sup>rd</sup> Edition, ISBN 9781718502703, published by No Starch Press Inc. 245 8<sup>th</sup> Street, San Francisco, California United States 94103.

The Polish-language 3<sup>rd</sup> edition Copyright © 2024 by Helion S.A. under license by No Starch Press Inc. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[https://helion.pl/user/opinie/pytip3\\_ebook](https://helion.pl/user/opinie/pytip3_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

## ***Opinie o książce***

„Doskonale się składa, że wydawnictwo No Starch Press zdecydowało się wydać przyszły klasyk, który powinien być stawiany obok bardziej tradycyjnych książek dotyczących programowania. *Python. Instrukcje dla programisty* jest jedną z takich książek.”

— GREG LADEN, SCIENCEBLOGS

„Książka zawiera omówienie dość skomplikowanych projektów, przedstawionych w spójny, logiczny i przyjemny sposób, który pozwala czytelnikowi skoncentrować się na danym zagadnieniu.”

— FULL CIRCLE MAGAZINE

„Doskonałe przedstawienie materiału z dobrym wyjaśnieniem przykładowych fragmentów kodu. Ta książka rozwija się wraz z czytelnikiem — poruszasz się małymi krokami do przodu, zaczynasz tworzyć coraz bardziej skomplikowany kod, a po drodze wszystko jest dokładnie wyjaśnione.”

— FLICKTHROUGH REVIEWS

„Poznawanie Pythona z książką *Python. Instrukcje dla programisty* było wyjątkowo przyjemnym doświadczeniem! To jest doskonała pozycja dla osób dopiero stawiających pierwsze kroki w Pythonie.”

— MIKKE GOES CODDING

„Książka doskonale spełnia swoje zadanie. Znajdziesz w niej dużo użytecznych ćwiczeń, a także trzy nieco bardziej wymagające i wciągające projekty.”

— REALPYTHON.COM

„Dość szybkie, choć jednocześnie obszerne wprowadzenie do programowania w Pythonie. To kolejna świetna książka do Twojej biblioteki, pomocna w ostatecznym opanowaniu Pythona.”

— TUTORIALEDGE.NET

„Kapitalna pozycja dla zupełnie początkujących, bez żadnego doświadczenia w programowaniu. Jeżeli szukasz solidnego i jasnego wprowadzenia do Pythona, gorąco polecam tę książkę.”

— WHATPIXEL.COM

„Znajdziesz w tej książce dosłownie wszystko, co powinieneś wiedzieć o Pythonie, a nawet jeszcze więcej.”

— FIREBEARSTUDIO.COM

„Wprawdzie w książce *Python. Instrukcje dla programisty* uczysz się tworzyć kod za pomocą Pythona, ale jednocześnie poznajesz podstawy programowania, które mają zastosowanie w większości języków.”

— GREAT LAKES GEEK

Dla mojego Taty, który zawsze znajdował czas,  
aby odpowiedzieć na moje pytania dotyczące programowania.  
Dla Evera, który dopiero zaczyna zadawać mi swoje pytania.

## O autorze

Eric Matthes przez 25 lat był nauczycielem matematyki i przedmiotów ścisłych w szkole średniej. Prowadził kursy Pythona dla początkujących, gdy tylko znalazł sposób na ich dopasowanie do programu nauczania. Obecnie Eric jest programistą zaangażowanym w rozwój wielu projektów open source. Mają one różnorodne cele — od pomocy w przewidywaniu osuwisk występujących w górach po ułatwianie procesu wdrażania projektów Django. W czasie wolnym uprawia wspinaczkę górską oraz spędza chwile z rodziną.

## O korektorze merytorycznym

Kenneth Love razem z rodziną i kotami mieszka na Wybrzeżu Północno-Zachodnim. Już od wielu lat jest nauczycielem, programistą Pythona i prelegentem na konferencjach. Ponadto współpracuje przy wielu projektach open source.

# Spis treści

WPROWADZENIE DO TRZECIEGO WYDANIA KSIĄŻKI .....	21
PODZIĘKOWANIA .....	24
WPROWADZENIE .....	26
<b>CZĘŚĆ I. PODSTAWY .....</b>	<b>31</b>
<b>1</b>	
<b>ROZPOCZĘCIE PRACY .....</b>	<b>33</b>
Przygotowanie środowiska programistycznego .....	33
Wersje Pythona .....	33
Wykonanie fragmentu kodu w Pythonie .....	34
Edytor tekstu VS Code .....	34
Python w różnych systemach operacyjnych .....	35
Python w systemie Windows .....	35
Python w systemie macOS .....	37
Python w systemach z rodziny Linux .....	39
Uruchomienie programu typu „Witaj, świeciel!” .....	40
Instalacja w VS Code rozszerzenia przeznaczonego do obsługi Pythona .....	40
Uruchomienie programu typu „Witaj, świeciel!” .....	41
Rozwiązywanie problemów podczas instalacji .....	42
Uruchamianie programów Pythona z poziomu powłoki .....	43
W systemie Windows .....	43
W systemach macOS i Linux .....	44
Podsumowanie .....	45
<b>2</b>	
<b>ZMIENNE I PROSTE TYPY DANYCH .....</b>	<b>46</b>
Co tak naprawdę dzieje się po uruchomieniu hello_world.py? .....	46
Zmienne .....	47
Nadawanie nazw zmiennym i używanie zmiennych .....	48
Unikanie błędów związanych z nazwami podczas używania zmiennych .....	49
Zmienna to etykieta .....	50

<b>Ciągi tekstowe .....</b>	<b>51</b>
Zmiana wielkości liter ciągu tekstowego za pomocą metod .....	51
Używanie zmiennych w ciągach tekstowych .....	53
Dodawanie białych znaków do ciągów tekstowych za pomocą tabulatora i znaku nowego wiersza .....	54
Usunięcie białych znaków .....	55
Usunięcie prefiksu .....	56
Unikanie błędów składni w ciągach tekstowych .....	57
<b>Liczby .....</b>	<b>59</b>
Liczby całkowite .....	59
Liczby zmiennoprzecinkowe .....	60
Liczby całkowite i zmiennoprzecinkowe .....	61
Znaki podkreślenia w liczbach .....	61
Wiele przypisań .....	62
Stałe .....	62
<b>Komentarze .....</b>	<b>63</b>
Jak można utworzyć komentarz? .....	63
Jakiego rodzaju komentarze należy tworzyć? .....	63
<b>Zen Pythona .....</b>	<b>64</b>
<b>Podsumowanie .....</b>	<b>66</b>
 <b>3</b>	
<b>WPROWADZENIE DO LIST .....</b>	<b>67</b>
Czym jest lista? .....	67
Uzyskanie dostępu do elementów listy .....	68
Numeracja indeksu zaczyna się od 0, a nie od 1 .....	68
Użycie poszczególnych wartości listy .....	69
Zmienianie, dodawanie i usuwanie elementów .....	70
Modyfikowanie elementów na liście .....	70
Dodawanie elementów do listy .....	71
Usuwanie elementu z listy .....	73
Organizacja listy .....	77
Trwałe sortowanie listy za pomocą metody sort() .....	78
Tymczasowe sortowanie listy za pomocą funkcji sorted() .....	78
Wyświetlanie listy w odwrotnej kolejności alfabetycznej .....	80
Określenie wielkości listy .....	80
Unikanie błędów indeksu podczas pracy z listą .....	81
Podsumowanie .....	83
 <b>4</b>	
<b>PRACA Z LISTĄ .....</b>	<b>84</b>
Iteracja przez całą listę .....	84
Dokładniejsza analiza pętli .....	85
Wykonanie większej liczby zadań w pętli for .....	86
Wykonywanie operacji po pętli for .....	88

Unikanie błędów związanych z wcięciami .....	89
Brak wcięcia .....	89
Brak wcięcia dodatkowych wierszy .....	90
Niepotrzebne wcięcie .....	90
Niepotrzebne wcięcie po pętli .....	91
Brak dwukropka .....	92
Tworzenie list liczbowych .....	93
Użycie funkcji range() .....	93
Użycie funkcji range() do utworzenia listy liczb .....	94
Proste dane statystyczne dotyczące listy liczb .....	96
Lista składana .....	96
Praca z fragmentami listy .....	98
Wycinek listy .....	98
Iteracja przez wycinek .....	100
Kopiowanie listy .....	101
Krotka .....	104
Definiowanie krotki .....	104
Iteracja przez wszystkie wartości krotki .....	105
Nadpisanie krotki .....	105
Styl tworzonego kodu .....	106
Konwencje stylu .....	107
Wcięcia .....	107
Długość wiersza .....	108
Puste wiersze .....	108
Inne specyfikacje stylu .....	108
Podsumowanie .....	109
<b>5 KONSTRUKCJA IF .....</b>	<b>110</b>
Prosty przykład .....	110
Test warunkowy .....	111
Sprawdzenie równości .....	111
Ignorowanie wielkości liter podczas sprawdzania równości .....	112
Sprawdzenie nierówności .....	113
Porównania liczbowe .....	114
Sprawdzanie wielu warunków .....	115
Sprawdzanie, czy wartość znajduje się na liście .....	116
Sprawdzanie, czy wartość nie znajduje się na liście .....	116
Wyrażenie boolowskie .....	117
Polecenie if .....	118
Proste polecenia if .....	118
Polecenia if-else .....	119
Konstrukcja if-elif-else .....	120
Użycie wielu bloków elif .....	122
Pominięcie bloku else .....	123
Sprawdzanie wielu warunków .....	123

Używanie polecen if z listami .....	126
Sprawdzanie pod kątem wartości specjalnych .....	126
Sprawdzanie, czy lista nie jest pusta .....	128
Użycie wielu list .....	129
Nadawanie stylu poleceniom if .....	131
Podsumowanie .....	132
<b>6</b>	
<b>SŁOWNIKI .....</b>	<b>133</b>
Prosty słownik .....	133
Praca ze słownikami .....	134
Uzyskiwanie dostępu do wartości słownika .....	135
Dodanie nowej pary klucz-wartość .....	136
Rozpoczęcie pracy od pustego słownika .....	137
Modyfikowanie wartości słownika .....	137
Usuwanie pary klucz-wartość .....	139
Słownik podobnych obiektów .....	139
Używanie metody get() w celu uzyskania dostępu do wartości .....	141
Iteracja przez słownik .....	143
Iteracja przez wszystkie pary klucz-wartość .....	143
Iteracja przez wszystkie klucze słownika .....	145
Iteracja przez uporządkowane klucze słownika .....	147
Iteracja przez wszystkie wartości słownika .....	148
Zagnieżdżanie .....	150
Lista słowników .....	150
Lista w słowniku .....	153
Słownik w słowniku .....	156
Podsumowanie .....	158
<b>7</b>	
<b>DANE WEJŚCIOWE UŻYTKOWNIKA I PĘTŁA WHILE .....</b>	<b>159</b>
Jak działa funkcja input()? .....	160
Przygotowanie jasnych i zrozumiałych komunikatów .....	160
Użycie funkcji int() do akceptowania liczbowych danych wejściowych .....	162
Operator modulo .....	163
Wprowadzenie do pętli while .....	164
Pętla while w działaniu .....	165
Umożliwienie użytkownikowi podjęcia decyzji o zakończeniu działania programu .....	165
Użycie flagi .....	167
Użycie polecenia break do opuszczenia pętli .....	169
Użycie polecenia continue w pętli .....	170
Unikanie pętli działającej w nieskończoność .....	170

Użycie pętli while wraz z listami i słownikami .....	172
Przenoszenie elementów z jednej listy na drugą .....	172
Usuwanie z listy wszystkich egzemplarzy określonej wartości .....	174
Umieszczenie w słowniku danych wejściowych wprowadzonych przez użytkownika .....	174
Podsumowanie .....	176
<b>8</b>	
<b>FUNKCJE .....</b>	<b>177</b>
Definiowanie funkcji .....	177
Przekazywanie informacji do funkcji .....	178
Argumenty i parametry .....	179
Przekazywanie argumentów .....	180
Argumenty pozycyjne .....	180
Argumenty w postaci słów kluczowych .....	182
Wartości domyślne .....	183
Odpowiedniki wywołań funkcji .....	184
Unikanie błędów związanych z argumentami .....	185
Wartość zwrotna .....	186
Zwrot prostej wartości .....	187
Definiowanie argumentu jako opcjonalnego .....	187
Zwrot słownika .....	189
Używanie funkcji wraz z pętlą while .....	190
Przekazywanie listy .....	193
Modyfikowanie listy w funkcji .....	193
Uniemożliwianie modyfikowania listy przez funkcję .....	196
Przekazywanie dowolnej liczby argumentów .....	197
Argumenty pozycyjne i przekazywanie dowolnej liczby argumentów .....	199
Używanie dowolnej liczby argumentów w postaci słów kluczowych .....	200
Przechowywanie funkcji w modułach .....	202
Import całego modułu .....	202
Import określonych funkcji .....	203
Użycie słowa kluczowego as w celu zdefiniowania aliasu funkcji .....	204
Użycie słowa kluczowego as w celu zdefiniowania aliasu modułu .....	205
Import wszystkich funkcji modułu .....	205
Nadawanie stylu funkcjom .....	206
Podsumowanie .....	207
<b>9</b>	
<b>KLASY .....</b>	<b>209</b>
Utworzenie i użycie klasy .....	210
Utworzenie klasy Dog .....	210
Metoda __init__() .....	211
Utworzenie egzemplarza na podstawie klasy .....	212

Praca z klasami i egzemplarzami .....	215
Klasa Car .....	215
Przypisanie atrybutowi wartości domyślnej .....	216
Modyfikacja wartości atrybutu .....	217
Dziedziczenie .....	221
Metoda <code>__init__()</code> w klasie potomnej .....	221
Definiowanie atrybutów i metod dla klasy potomnej .....	223
Nadpisywanie metod klasy nadzędnej .....	225
Egzemplarz jako atrybut .....	225
Modelowanie rzeczywistych obiektów .....	228
Import klas .....	229
Import pojedynczej klasy .....	229
Przechowywanie wielu klas w module .....	231
Import wielu klas z modułu .....	232
Import całego modułu .....	233
Import wszystkich klas z modułu .....	234
Import modułu w module .....	234
Używanie aliasów .....	235
Określenie swojego sposobu pracy .....	236
Biblioteka standardowa Pythona .....	237
Nadawanie stylu klasom .....	238
Podsumowanie .....	239
<b>10</b>	
<b>PLIKI I WYJĄTKI .....</b>	<b>240</b>
Odczytywanie danych z pliku .....	241
Wczytywanie całego pliku .....	241
Względna i bezwzględna ścieżka dostępu do pliku .....	243
Odczytywanie wiersz po wierszu .....	244
Praca z zawartością pliku .....	245
Ogromne pliki, czyli na przykład milion cyfr .....	246
Czy data Twoich urodzin znajduje się w liczbie pi? .....	247
Zapisywanie danych w pliku .....	249
Zapisywanie pojedynczego wiersza .....	249
Zapisywanie wielu wierszy .....	249
Wyjątki .....	251
Obsługiwianie wyjątku <code>ZeroDivisionError</code> .....	251
Używanie bloku <code>try-except</code> .....	252
Używanie wyjątków w celu uniknięcia awarii programu .....	252
Blok <code>else</code> .....	253
Obsługa wyjątku <code>FileNotFoundException</code> .....	255
Analiza tekstu .....	256
Praca z wieloma plikami .....	257
Ciche niepowodzenie .....	259
Które błędy należy zgłaszać? .....	260

Przechowywanie danych .....	261
Używanie json.dumps() i json.loads() .....	262
Zapisywanie i odczytywanie danych wygenerowanych przez użytkownika .....	263
Refaktoryzacja .....	266
Podsumowanie .....	269
<b>11 TESTOWANIE KODU .....</b>	<b>270</b>
Instalowanie pytest za pomocą pip .....	271
Aktualnianie pip .....	271
Instalowanie pytest .....	272
Testowanie funkcji .....	272
Test jednostkowy i zestaw testów .....	274
Zaliczenie testu .....	274
Wykonywanie testu .....	275
Niezaliczenie testu .....	276
Reakcja na niezaliczony test .....	277
Dodanie nowego testu .....	279
Testowanie klasy .....	280
Różne rodzaje metod assert .....	280
Klasa do przetestowania .....	281
Testowanie klasy AnonymousSurvey .....	283
Używanie danych testowych .....	285
Podsumowanie .....	287
<b>CZĘŚĆ II. PROJEKTY .....</b>	<b>289</b>
<b>12 STATEK, KTÓRY STRZELA POCISKAMI .....</b>	<b>291</b>
Planowanie projektu .....	292
Instalacja Pygame .....	292
Rozpoczęcie pracy nad projektem gry .....	293
Utworzenie okna Pygame i reagowanie na działania użytkownika .....	293
Określenie liczby klatek na sekundę .....	295
Zdefiniowanie koloru tła .....	296
Utworzenie klasy ustawień .....	297
Dodanie obrazu statku kosmicznego .....	298
Utworzenie klasy statku kosmicznego .....	299
Wyświetlenie statku kosmicznego na ekranie .....	301
Refaktoryzacja, czyli metody _check_events() i _update_screen() .....	302
Metoda _check_events() .....	303
Metoda _update_screen() .....	303

Kierowanie statkiem kosmicznym .....	304
Reakcja na naciśnięcie klawisza .....	305
Umożliwienie nieustanego ruchu .....	305
Poruszanie statkiem w obu kierunkach .....	307
Dostosowanie szybkości statku .....	309
Ograniczenie zasięgu poruszania się statku .....	310
Refaktoryzacja metody <code>_check_events()</code> .....	311
Naciśnięcie klawisza Q w celu zakończenia gry .....	312
Uruchamianie gry w trybie pełnoekranowym .....	312
Krótkie powtóżenie .....	313
<code>alien_invasion.py</code> .....	313
<code>settings.py</code> .....	314
<code>ship.py</code> .....	314
Wystrzeliwanie pocisków .....	315
Dodawanie ustawień dotyczących pocisków .....	315
Utworzenie klasy Bullet .....	315
Przechowywanie pocisków w grupie .....	317
Wystrzeliwanie pocisków .....	318
Usuwanie niewidocznych pocisków .....	320
Ograniczenie liczby pocisków .....	321
Utworzenie metody <code>_update_bullets()</code> .....	321
Podsumowanie .....	323
<b>13</b>	
<b>OBCY! .....</b>	<b>324</b>
Przegląd projektu .....	325
Utworzenie pierwszego obcego .....	325
Utworzenie klasy Alien .....	326
Utworzenie egzemplarza obcego .....	327
Utworzenie floty obcych .....	328
Utworzenie rzędów obcych .....	329
Refaktoryzacja metody <code>_create_fleet()</code> .....	330
Dodawanie rzędów .....	332
Poruszanie flotą obcych .....	334
Przesunięcie obcych w prawo .....	334
Zdefiniowanie ustawień dla kierunku poruszania się floty .....	336
Sprawdzenie, czy obcy dotarł do krawędzi ekranu .....	336
Przesunięcie floty w dół i zmiana kierunku .....	337
Zestrzelianie obcych .....	339
Wykrywanie kolizji z pociskiem .....	339
Utworzenie większych pocisków w celach testowych .....	340
Ponowne utworzenie floty .....	340
Zwiększenie szybkości pocisku .....	342
Refaktoryzacja metody <code>_update_bullets()</code> .....	342

Zakończenie gry .....	343
Wykrywanie kolizji między obcym i statkiem .....	343
Reakcja na kolizję między obcym i statkiem .....	344
Obcy, który dociera do dolnej krawędzi ekranu .....	347
Koniec gry! .....	348
Ustalenie, które komponenty gry powinny być uruchomione .....	349
Podsumowanie .....	350
<b>14</b>	
<b>PUNKTACJA .....</b>	<b>351</b>
Dodanie przycisku Gra .....	351
Utworzenie klasy Button .....	352
Wyświetlenie przycisku na ekranie .....	354
Uruchomienie gry .....	355
Zerowanie gry .....	356
Dezaktywacja przycisku Gra .....	357
Ukrycie kurSORA myszy .....	357
Zmiana poziomu trudności .....	358
Zmiana ustawień dotyczących szybkości .....	359
Wyzerowanie szybkości .....	360
Punktacja .....	361
Wyświetlanie punktacji .....	362
Utworzenie tablicy wyników .....	363
Aktualnienie punktacji po zestrzeleniu obcego .....	364
Zerowanie wyniku .....	366
Zagwarantowanie uwzględnienia wszystkich trafień .....	366
Zwiększenie liczby zdobywanych punktów .....	367
Zaokrąglanie punktacji .....	368
Najlepsze wyniki .....	369
Wyświetlenie aktualnego poziomu gry .....	371
Wyświetlenie liczby statków .....	375
Podsumowanie .....	378
<b>15</b>	
<b>GENEROWANIE DANYCH .....</b>	<b>379</b>
Instalacja matplotlib .....	380
Wygenerowanie prostego wykresu liniowego .....	380
Zmienianie etykiety i grubości wykresu .....	381
Poprawianie wykresu .....	383
Używanie wbudowanych stylów .....	384
Używanie funkcji scatter() do wyświetlania poszczególnych punktów i nadawania im stylu .....	386
Wyświetlanie serii punktów za pomocą funkcji scatter() .....	387
Automatyczne obliczanie danych .....	388
Dostosowanie znaczników osi do własnych potrzeb .....	389

Definiowanie własnych kolorów .....	390
Użycie mapy kolorów .....	390
Automatyczny zapis wykresu .....	392
<b>Błędzenie losowe .....</b>	<b>392</b>
Utworzenie klasy RandomWalk .....	392
Wybór kierunku .....	393
Wyświetlenie wykresu błędzenia losowego .....	394
Wygenerowanie wielu błędzeń losowych .....	395
Nadawanie stylu danym wygenerowanym przez błędzenie losowe .....	396
<b>Symulacja rzutu kością do gry za pomocą plotly .....</b>	<b>402</b>
Instalacja plotly .....	402
Utworzenie klasy Die .....	403
Rzut kością do gry .....	403
Analiza wyników .....	404
Utworzenie histogramu .....	405
Dostosowanie wykresu do własnych potrzeb .....	406
Rzut dwiema kości mi .....	407
Dalsze dostosowanie wykresu do własnych potrzeb .....	409
Rzut kości mi o różnej liczbie ścianek .....	410
Zapisywanie wykresu .....	411
<b>Podsumowanie .....</b>	<b>412</b>
<b>16 POBIERANIE DANYCH .....</b>	<b>413</b>
<b>Format CSV .....</b>	<b>414</b>
Przetwarzanie nagłówków pliku CSV .....	414
Wyświetlanie nagłówków i ich położenia .....	415
Wyodrębnienie i odczytanie danych .....	416
Wyświetlenie danych na wykresie temperatury .....	417
Moduł datetime .....	418
Wyświetlanie daty .....	419
Wyświetlenie dłuższego przedziału czasu .....	420
Wyświetlenie drugiej serii danych .....	421
Nakładanie cienia na wykresie .....	423
Sprawdzenie pod kątem błędów .....	424
Samodzielne pobieranie danych .....	428
<b>Mapowanie globalnych zbiorów danych – format GeoJSON .....</b>	<b>429</b>
Pobranie danych dotyczących trzęsień ziemi .....	430
Analiza danych GeoJSON .....	430
Utworzenie listy trzęsień ziemi .....	433
Wyodrębnienie siły trzęsienia ziemi .....	433
Wyodrębnienie danych o miejscu wystąpienia trzęsienia ziemi .....	434
Budowanie mapy świata .....	435
Przedstawienie siły trzęsienia ziemi .....	436

Dostosowanie koloru punktu .....	437
Inne skale kolorów .....	439
Dodanie tekstu wyświetlanego po wskazaniu punktu na mapie .....	439
Podsumowanie .....	441
<b>17</b>	
<b>PRACA Z API .....</b>	<b>442</b>
Użycie API .....	442
Git i GitHub .....	443
Żądanie danych za pomocą wywołania API .....	443
Instalacja requests .....	444
Przetworzenie odpowiedzi API .....	445
Praca ze słownikiem odpowiedzi .....	446
Podsumowanie repozytoriów najczęściej oznaczanych gwiazdką .....	449
Monitorowanie ograniczeń liczby wywołań API .....	450
Wizualizacja repozytoriów za pomocą pakietu plotly .....	451
Nadawanie stylu wykresowi .....	452
Dodanie własnych podpowiedzi .....	454
Dodawanie łączy do wykresu .....	455
Dostosowanie kolorów znaczników do własnych potrzeb .....	456
Więcej o plotly i API GitHub .....	457
Hacker News API .....	457
Podsumowanie .....	462
<b>18</b>	
<b>ROZPOCZĘCIE PRACY Z DJANGO .....</b>	<b>463</b>
Przygotowanie projektu .....	464
Opracowanie specyfikacji .....	464
Utworzenie środowiska wirtualnego .....	464
Aktywacja środowiska wirtualnego .....	465
Instalacja frameworka Django .....	465
Utworzenie projektu w Django .....	466
Utworzenie bazy danych .....	467
Przegląd projektu .....	468
Uruchomienie aplikacji .....	469
Definiowanie modeli .....	470
Aktywacja modeli .....	471
Witryna administracyjna Django .....	473
Zdefiniowanie modelu Entry .....	476
Migracja modelu Entry .....	477
Rejestracja modelu Entry w witrynie administracyjnej .....	478
Powłoka Django .....	479
Tworzenie stron internetowych — strona główna aplikacji .....	481
Mapowanie adresu URL .....	482
Utworzenie widoku .....	484
Utworzenie szablonu .....	484

Utworzenie dodatkowych stron .....	486
Dziedziczenie szablonu .....	486
Strona tematów .....	489
Strony poszczególnych tematów .....	492
Podsumowanie .....	496
<b>19 KONTA UŻYTKOWNIKÓW .....</b>	<b>497</b>
Umożliwienie użytkownikom wprowadzania danych .....	497
Dodawanie nowego tematu .....	498
Dodawanie nowych wpisów .....	503
Edycja wpisu .....	507
Konfiguracja kont użytkowników .....	511
Aplikacja accounts .....	511
Strona logowania .....	512
Wylogowanie .....	515
Strona rejestracji użytkownika .....	516
Umożliwienie użytkownikom bycia właścicielami swoich danych .....	520
Ograniczenie dostępu za pomocą dekoratora @login_required .....	520
Powiązanie danych z określonymi użytkownikami .....	522
Przyznanie dostępu jedynie odpowiednim użytkownikom .....	525
Ochrona tematów użytkownika .....	526
Ochrona strony edit_entry .....	527
Powiązanie nowego tematu z bieżącym użytkownikiem .....	528
Podsumowanie .....	529
<b>20 NADANIE STYLU I WDROŻENIE APLIKACJI .....</b>	<b>531</b>
Nadanie stylu aplikacji Learning Log .....	532
Aplikacja django-bootstrap5 .....	532
Użycie Bootstrapa do nadania stylu aplikacji Learning Log .....	533
Modyfikacja pliku base.html .....	533
Użycie elementu Jumbotron do nadania stylu stronie głównej .....	540
Nadanie stylu stronie logowania .....	541
Nadanie stylu stronie tematów .....	542
Nadanie stylów wpisom na stronie tematu .....	543
Wdrożenie aplikacji Learning Log .....	545
Utworzenie konta w Platform.sh .....	546
Instalacja Platform.sh CLI .....	546
Instalacja platformshconfig .....	546
Utworzenie pliku requirements.txt .....	547
Dodatkowe wymagania dotyczące wdrożenia .....	547
Dodawanie plików konfiguracyjnych .....	548
Modyfikacja pliku settings.py dla Platform.sh .....	552
Użycie Gita do monitorowania plików projektu .....	553

Utworzenie projektu w Platform.sh .....	555
Przekazanie projektu do Platform.sh .....	557
Wyświetlenie wdrożonego projektu .....	558
Dopracowanie wdrożenia projektu w Platform.sh .....	558
Utworzenie własnych stron błędu .....	561
Nieustanna rozbudowa .....	563
Usunięcie projektu z Platform.sh .....	564
Podsumowanie .....	565
<b>A</b>	
<b>INSTALACJA PYTHONA I ROZWIĄZYWANIE PROBLEMÓW .....</b>	<b>567</b>
Python w Windows .....	567
Użycie polecenia py zamiast python .....	568
Ponowna instalacja Pythona .....	568
Python w systemie macOS .....	568
Przypadkowa instalacja wersji dostarczanej przez Apple .....	569
Python 2 w starszych wydaniach systemu macOS .....	569
Python w systemie Linux .....	569
Używanie domyślnej instalacji Pythona .....	569
Instalacja najnowszej wersji Pythona .....	570
Sprawdzenie aktualnie używanej wersji Pythona .....	570
Słowa kluczowe Pythona i wbudowane funkcje .....	571
Słowa kluczowe Pythona .....	571
Wbudowane funkcje Pythona .....	571
<b>B</b>	
<b>EDYTOR Y TEKSTU I ŚRODOWISKA IDE .....</b>	<b>573</b>
Efektywna praca z VS Code .....	574
Konfigurowanie VS Code .....	575
Wybrane skróty klawiszowe VS Code .....	577
Inne edytory tekstu i środowiska IDE .....	579
IDLE .....	579
Geany .....	579
Sublime Text .....	579
Emacs i vim .....	579
PyCharm .....	580
Notatniki Jupyter Notebooks .....	580
<b>C</b>	
<b>UZYSKIWANIE POMOCY .....</b>	<b>581</b>
Pierwsze kroki .....	581
Spróbuj jeszcze raz .....	582
Chwila odpoczynku .....	582
Korzystaj z zasobów tej książki .....	583

Wyszukiwanie informacji w internecie .....	583
Stack Overflow .....	583
Oficjalna dokumentacja Pythona .....	584
Oficjalna dokumentacja biblioteki .....	584
r/learnpython .....	584
Posty na blogach .....	585
Discord .....	585
Slack .....	585
<b>D</b>	
<b>UŻYWANIE GIT A DO KONTROLI WERSJI .....</b>	<b>586</b>
Instalacja Gita .....	587
Konfiguracja Gita .....	587
Tworzenie projektu .....	588
Ignorowanie plików .....	588
Inicjalizacja repozytorium .....	589
Sprawdzanie stanu .....	589
Dodawanie plików do repozytorium .....	590
Zatwierdzanie plików .....	590
Sprawdzanie dziennika projektu .....	591
Drugie zatwierdzenie .....	591
Przywracanie stanu projektu .....	593
Przywracanie projektu do poprzedniego stanu .....	594
Usuwanie repozytorium .....	596
<b>E</b>	
<b>ROZWIĄZYWANIE PROBLEMÓW Z WDROŻENIAMI .....</b>	<b>598</b>
Jak wygląda proces wdrażania? .....	598
Podstawy rozwiązywania problemów .....	599
Kierowanie się podpowiedziami wyświetlanymi na ekranie .....	600
Odczytywanie danych wyjściowych dzienników zdarzeń .....	601
Rozwiązywanie problemów dotyczących konkretnego systemu operacyjnego .....	603
Wdrażanie z poziomu systemu Windows .....	604
Wdrażanie z poziomu systemu macOS .....	605
Wdrażanie z poziomu systemu Linux .....	606
Inne podejścia w zakresie wdrożenia .....	607

# Wprowadzenie do trzeciego wydania książki

Reakcja na pierwsze i drugie wydania książki *PYTHON. INSTRUKCJE DLA PROGRAMISTY* była w ogromnej mierze pozytywna. Calkowity nakład książki, wraz z jej tłumaczeniami na inne języki, wyniósł ponad milion egzemplarzy. Otrzymałem wiele listów i wiadomości e-mail od czytelników — z jednej strony najmłodszy z nich miał zaledwie 10 lat, z drugiej zaś pisali do mnie również emeryci — którzy w wolnym czasie chcieli zająć się programowaniem. Okazało się, że moja książka jest wykorzystywana w szkołach średnich i wyższych. Uczniowie korzystający z bardziej zaawansowanych pozycji uznawali moją za warte uwagi uzupełnienie. Wielu osobom ta książka pozwoliła rozbudować umiejętności potrzebne w pracy i pomogła rozpocząć prace nad własnymi projektami. Ujmując rzecz najkrócej, książka znalazła wiele zastosowań, do których ją napisałem.

Możliwość przygotowania trzeciego wydania była prawdziwą przyjemnością. Wprawdzie Python zalicza się do dojrzałych języków programowania, ale podobnie jak inne języki jest nieustannie rozwijany. Moim celem było, aby ta książka pozostała świetnym źródłem wprowadzającym do programowania w Pythonie. Dzięki jej lekturze dowiesz się wszystkiego, co jest potrzebne, aby rozpocząć pracę nad własnymi projektami. Ponadto zdobędziesz solidne podstawy pomocne w dalszej nauce Pythona. Uaktualnilem wybrane sekcje, aby odzwierciedlić nowe, prostsze sposoby na wykonywanie pewnych zadań w Pythonie. Postarałem się też wyjaśnić niektóre kwestie w miejscach, gdzie szczegóły dotyczące języka nie zostały przedstawione wystarczająco jasno. Wszystkie projekty całkowicie uaktualnilem z wykorzystaniem popularnych, doskonale opracowanych bibliotek, których możesz użyć, tworząc własne projekty.

Oto podsumowanie zmian wprowadzonych w trzecim wydaniu książki:

- W rozdziale 1. przedstawiłem informacje na temat edytora tekstu VS Code, który jest popularny zarówno wśród początkujących, jak i profesjonalnych programistów oraz doskonale działa na wszystkich najważniejszych platformach.
- W rozdziale 2. omówiłem nowe metody `removeprefix()` i `removesuffix()`, które okazują się użyteczne w pracy z plikami i adresami URL. Znajdziesz tutaj także informacje na temat usprawnionych w Pythonie komunikatów błędów, które obecnie są znacznie dokładniejsze i pomagają rozwiązywać problemy z kodem, gdy coś nie działa zgodnie z oczekiwaniemi.
- W rozdziale 10. użyłem modułu `pathlib` do pracy z plikami. To znacznie prostsze podejście podczas odczytywania i zapisywania plików.
- W rozdziale 11. do tworzenia zautomatyzowanych testów dla kodu źródłowego wykorzystałem bibliotekę `pytest`, która stała się standardowym narzędziem przeznaczonym do tworzenia testów w Pythonie. Okazuje się wystarczająco przyjazna użytkownikowi, aby można było używać jej do przygotowywania pierwszych testów. Jeżeli chcesz zostać programistą Pythona, będziesz z niej korzystać w swojej karierze.
- W projekcie *Inwazja obcych* (rozdziały od 12. do 14.) wykorzystałem ustawienie kontrolujące liczbę klatek wyświetlanego na sekundę. Dzięki temu gra działa spójnie w różnych systemach operacyjnych. Zastosowałem także prostsze podejście podczas tworzenia floty obcych. Ponadto poprawiłem ogólną organizację projektu i go uporządkowałem.
- W projektach dotyczących wizualizacji danych (rozdziały od 15. do 17.) użyłem najnowszych wersji pakietów `matplotlib` i `plotly`. W wizualizacjach tworzonych za pomocą `matplotlib` zostały wykorzystane uaktualnione ustawienia stylów. W projekcie błądzenia losowego wprowadziłem drobne usprawnienia poprawiające dokładność wykresów. Dzięki temu można dostrzec większą różnorodność wzorców podczas generowania nowych danych. We wszystkich projektach `plotly` wykorzystałem moduł `plotly express`, który umożliwia wygenerowanie początkowych wizualizacji za pomocą dosłownie kilku wierszy kodu. Bardzo łatwo można zapoznać się z różnymi wizualizacjami przed wybraniem ich dowolnego rodzaju, a następnie skoncentrować się na dopracowaniu poszczególnych elementów wybranego typu wykresu.
- Projekt przedstawiony w rozdziałach od 18. do 20. został oparty na najnowszych wersjach Django i korzysta ze stylów wprowadzonych w najnowszej wersji framework'a Bootstrap. Nazwy niektórych elementów projektu zostały zmienione, aby ułatwić stosowanie się do jego ogólnej organizacji. Obecnie projekt został wdrożony do `Platform.sh`, czyli nowoczesnej usługi hostingowej dla projektów Django. Proces wdrożenia jest kontrolowany za pomocą plików konfiguracyjnych `YAML`, co zapewnia doskonały nadzór nad wdrożeniem projektu. Takie podejście jest spójne z tym, jak zawodowi programiści wdrażają nowoczesne projekty Django.

- Uaktualniłem w pełni dodatek A, aby odzwierciedlał aktualnie najlepsze praktyki w zakresie instalacji Pythona w większości systemów operacyjnych. W dodatku B znajdziesz dość dokładne informacje o przygotowaniu edytora VS Code do pracy, a także informacje o obecnie najważniejszych edytorach tekstu i środowiskach IDE. W dodatku C skierowałem czytelników do nowszych i popularniejszych zasobów internetowych, w których można szukać pomocy w przypadku problemów z kodem Pythona. Natomiast dodatek D wciąż zawiera miniwprowadzenie do pracy z systemem kontroli wersji Git. Dodatek E jest nowością w trzecim wydaniu książki. Pomimo dobrych informacji dotyczących wdrażania tworzonych aplikacji nadal coś może pójść nie tak. W tym dodatku znajdziesz szczegółowe wskazówki na temat rozwiązywania problemów, przydatne, gdy proces wdrażania nie działa zgodnie z oczekiwaniemi.
- Skorowidz został uaktualniony, aby książkę można było wykorzystać w charakterze przewodnika dla Twoich wszystkich przyszłych projektów Pythona.

Dziękuję za lekturę książki *Python. Instrukcje dla programisty*. Jeżeli masz jakiekolwiek uwagi lub pytania, możesz się ze mną skontaktować — znajdziesz mnie w serwisie Twitter (@ehmatthes).

# Podziękowania

Powstanie tej książki byłoby niemożliwe bez wspaniałego i wyjątkowo profesjonalnego zespołu wydawnictwa No Starch Press. Bill Pollock zaproponował mi napisanie książki i jestem mu ogromnie wdzięczny za tę propozycję. Liz Chadwick pracowała nad wszystkimi trzema wydaniami i dzięki jej nieustającemu zaangażowaniu książka staje się coraz lepsza. Eva Morrow spojrzała świeżym okiem na nowe wydanie, a jej opinie również pomogły w ulepszeniu książki. Doceniam pomoc Douga McNaira w użyciu właściwego języka bez popadania w zbyt formalny styl. Jennifer Kepler nadzorowała proces powstawania kompletnej książki i cierpliwie pomagała przekształcić moją pracę w starannie dopracowany produkt.

W wydawnictwie No Starch Press jest wiele osób, dzięki którym ta książka osiągnęła sukces, a z którymi nie miałem okazji bezpośrednio współpracować. To wydawnictwo ma fantastyczny dział marketingu, którego praca nie ogranicza się do sprzedaży książek — ten zespół stara się, aby czytelnicy otrzymywali książki, których potrzebują, by zrealizować swoje cele. Wydawnictwo No Starch ma również świetny dział zajmujący się prawami autorskimi dla książek wydawanych w innych językach. Dzięki pracy tego działu moja książka została przetłumaczona na wiele języków i trafiła do czytelników z całego świata. Składam ogromne podziękowania wszystkim tym osobom, z którymi nie miałem możliwości bezpośrednio współpracować, a które pomogły mojej książce trafić do wielu czytelników.

Chciałbym w tym miejscu podziękować Kennethowi Love'owi za przeprowadzenie korekty merytorycznej wszystkich trzech wydań. Kennetha spotkałem kiedyś roku na konferencji PyCon — zarówno jego entuzjazm dotyczący języka, jak i sama społeczność Pythona okazały się dla mnie nieustannym źródłem inspiracji. Kenneth, jak zwykle, nie ograniczył się do prostego sprawdzenia faktów, ale

zaproponował także pewne zmiany, które mają pomóc początkującym programistom zdobyć solidną wiedzę w zakresie programowania w języku Python oraz ogólnie programowania. Zwrócił także uwagę na fragmenty, które sprawdzają się dobrze, ale można było je ulepszyć w nowym wydaniu książki. Wszelkie nieścisłości obecne w tej książce wynikają zatem jedynie z moich błędów.

Chciałbym podziękować wszystkim czytelnikom, którzy podzielili się swoimi wrażeniami po lekturze książki. Poznawanie podstaw programowania może zmienić sposób postrzegania świata, a czasami także mieć ogromny wpływ na innych. Uważnie zapoznaję się z takimi historiami i doceniam każdego, kto tak otwarcie podzielił się swoimi doświadczeniami.

Chciałbym również podziękować mojemu ojcu za wprowadzanie mnie do programowania już od najmłodszych lat, bez obaw, że uszkodzę jego sprzęt. Podziękowania składam także mojej żonie Erin za nieustanne wspieranie mnie i dodawanie mi odwagi podczas pisania tej książki oraz pracy nad wszystkimi jej wydaniami. Dziękuję też mojemu synowi Everowi, którego ciekawość świata inspiruje mnie każdego dnia.

# Wprowadzenie



KAŻDY PROGRAMISTA MA SWOJĄ HISTORIĘ DOTYCZĄCĄ UTWORZENIA PIERWSZEGO PROGRAMU. SAM NAUKĘ PROGRAMOWANIA ROZPOCZĄŁEM JUŻ JAKO DZIECKO, GDY MÓJ OJCIEC PRACOWAŁ DLA DIGITAL EQUIPMENT Corporation, czyli jednej z pionierskich firm ery nowoczesnych komputerów. Mój pierwszy program stworzyłem za pomocą komputera, który ojciec złożył w piwnicy. Ten komputer składał się z pozbawionej obudowy płyty głównej, do której była podłączona klawiatura oraz monitor kineskopowy. Utworzony przeze mnie program to była prosta gra — zadanie gracza polegało na odgadnięciu liczby. Oto przykład sesji z tej gry:

---

Mam na myśli pewną liczbę! Spróbuj ją odgadnąć: **25**

Za mała! Spróbuj ponownie: **50**

Za duża! Spróbuj ponownie: **42**

Dokładnie ta! Czy chcesz zagrać ponownie? (tak/nie) **nie**

Dziękuję za wspólną zabawę!

---

Pamiętam, jak z prawdziwą satysfakcją obserwowałem moją rodzinę grającą w utworzoną przeze mnie grę, która działała zgodnie z oczekiwaniemi.

Tego rodzaju wczesne doświadczenia mają na nas ogromny wpływ i pozostają w pamięci na zawsze. Rzeczywistą satysfakcję przynosi zbudowanie oprogramowania, które będzie przeznaczone do rozwiązywania konkretnych problemów i faktycznie będzie używane przez innych. Od oprogramowania, które teraz tworzę, wymagam znacznie więcej niż od programów, które budowałem w dzieciństwie. Jednak satysfakcja, którą odczuwam z powodu zaprojektowania funkcjonującej aplikacji, pozostała praktycznie taka sama.

# Do kogo jest skierowana ta książka?

Celem tej książki jest to, by jak najszybciej nauczyć Cię programowania w języku Python, abyś mógł tworzyć aplikacje działające zgodnie z oczekiwaniami — gry, wizualizacje danych i aplikacje internetowe. Jednocześnie chciałbym pomóc Ci w opanowaniu tych podstaw programowania, które będą Ci służyć do końca życia. Niniejsza książka została napisana dla Czytelników w dowolnym wieku, którzy nigdy nie programowali w języku Python lub w ogóle nie zajmowali się programowaniem. Jeżeli chcesz szybko opanować podstawy programowania, aby skoncentrować się na interesujących Cię projektach, a także jeśli chcesz sprawdzić wiedzę z zakresu nowych koncepcji, rozwiązyując konkretne problemy, ta książka jest skierowana właśnie do Ciebie. To także doskonały materiał dydaktyczny dla nauczycieli szkół średnich i wyższych, który swoim uczniom chętnie zaoferowałby aparte na projektach wprowadzenie do programowania. Jeżeli jesteś studentem uczelni i szukasz znacznie przystępniejszego wprowadzenia do języka Python niż w podręcznikach, niniejsza książka również jest dla Ciebie. Jeżeli chcesz zmienić zawód, niniejsza książka może Ci pomóc w przejściu na znacznie bardziej satysfakcjonującą ścieżkę kariery zawodowej. Ta pozycja sprawdziła się już w przypadku wielu czytelników, którym przyświecały różne cele.

## Czego nauczysz się z tej książki?

Materiał przedstawiony w książce ma Ci pomóc stać się dobrym programistą, zwłaszcza w zakresie języka Python. Poznasz dobre nawyki w programowaniu oraz solidne podstawy dotyczące ogólnych koncepcji w nim stosowanych. Po opanowaniu materiału przedstawionego w książce będziesz gotów przejść do doskonalenia się w użyciu bardziej zaawansowanych technik Pythona. Jeżeli zdecydujesz się na naukę innego języka programowania, to znajomość Pythona i ogólnych koncepcji programowania na pewno znacznie ułatwi Ci to zadanie.

W części pierwszej książki przedstawię podstawowe koncepcje programowania, które musisz znać, jeśli chcesz tworzyć programy w Pythonie. Te koncepcje są praktycznie takie same jak te, które poznajesz, gdy rozpoczynasz naukę dowolnego języka programowania. Zobaczysz różne rodzaje danych i poznasz sposoby ich przechowywania na zdefiniowanych w programach listach oraz w słownikach. Nauczysz się budować kolekcje danych i pracować z nimi w efektywny sposób. Zobaczysz, jak używać pętli `while` i konstrukcji `if` do testowania określonych warunków, które pozwalają wykonywać konkretne bloki kodu w zależności od wyniku testu. Możliwość wykonania jednego bloku kodu, gdy warunek przyjmuje wartość `True`, i innego, gdy warunek przyjmuje wartość `False`, znacząco pomaga w zautomatyzowaniu procesów.

Poznasz sposoby akceptowania pochodzących od użytkowników danych wejściowych, dzięki którym programy stają się bardziej interaktywne i mogą działać tak długo, dopóki użytkownik pozostaje aktywny. Zobaczysz, jak tworzyć funkcje pozwalające na wielokrotne wykorzystywanie fragmentów programów. Dzięki

temu blok kodu odpowiedzialny za wykonywanie określonego zadania będziesz musiał utworzyć tylko za pierwszym razem — później będziesz mógł z niego wielokrotnie korzystać. Następnie tę koncepcję rozszerzysz o bardziej zaawansowane zachowania oparte na klasach, co pozwoli całkiem prostemu programowi reagować w wielu różnych sytuacjach. Zobaczysz również, jak należy tworzyć programy, które w elegancki sposób będą obsługiwały najczęściej występujące błędy. Po przeanalizowaniu każdej z wymienionych podstawowych koncepcji stworzysz kilka krótkich programów przeznaczonych do rozwiązywania pewnych doskonale Ci znanych problemów. Później wykonasz pierwszy krok do średnio-zaawansowanego programowania, czyli dowiesz się, jak tworzyć testy dla kodu. Dzięki testom możesz rozbudowywać programy bez obaw, że zostaną do nich wprowadzone błędy. Wszystkie informacje przedstawione w części pierwszej książki mają przygotować Cię do zajęcia się większymi i bardziej skomplikowanymi projektami.

W części drugiej książki wiedzę zdobytą w części pierwszej wykorzystasz do realizacji trzech projektów. Możesz wykonać wszystkie trzy projekty lub tylko ten, który najbardziej Cię zainteresuje. W pierwszym projekcie (rozdziały od 12. do 14.) zajmiemy się budowaniem gry *Inwazja obcych*, czyli innej wersji klasycznej i dość popularnej gry zręcznościowej *Alien Invasion*, w której trudność rozgrywki rośnie wraz z postępem poczynionym przez gracza. Ukończenie tego projektu powinno dać Ci dobre podstawy do opracowywania własnych gier 2D. Nawet jeśli nie planujesz zostać programistą gier, praca nad tym projektem to przyjemny sposób na utrwalenie wiedzy i koncepcji poznanych w części pierwszej książki.

Drugi projekt (rozdziały od 15. do 17.) jest wprowadzeniem do wizualizacji danych. Naukowcy zajmujący się danymi potrafią ogromne ilości zebranych informacji wykorzystać, przygotowując wizualizacje za pomocą różnego rodzaju technik. Nauczysz się pracy ze zbiorami danych — zarówno tymi generowanymi przy użyciu kodu, jak i tymi pobieranymi z zasobów znajdujących się w internecie. Zobaczysz również, jak automatycznie pobierać dane z poziomu programów. Po skończeniu tego projektu będziesz miał wiedzę pozwalającą na tworzenie programów, które wykorzystują ogromne zbiory danych oraz przygotowują wizualizacje na podstawie tych przechowywanych informacji.

W trzecim projekcie (rozdziały od 18. do 20.) zbudujemy małą aplikację internetową o nazwie *Learning Log*. Ten projekt umożliwi użytkownikom prowadzenie czegoś w rodzaju dziennika dla pomysłów i koncepcji pojawiających się podczas poznawania określonego zagadnienia. Aplikacja pozwoli użytkownikom przechowywać oddzielnie dane dla poszczególnych tematów oraz składać konta, a tym samym rozpocząć tworzenie własnych dzienników. Dowiesz się, jak wdrożyć aplikację w serwerze WWW, aby mógł z niej korzystać każdy, kto tylko ma dostęp do internetu.

# Zasoby w internecie

Zasoby uzupełniające materiał zamieszczony w książce znajdziesz w internecie pod adresami <https://nostarch.com/python-crash-course-3rd-edition> i [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Te zasoby obejmują m.in.:

**Informacje dotyczące konfiguracji.** Wprawdzie przedstawione tutaj informacje są identyczne jak w książce, ale zawierają aktywne łącza pozwalające łatwo przechodzić do różnych zasobów. Jeżeli napotkasz jakiekolwiek trudności podeczas przygotowywania środowiska pracy, koniecznie zajrzyj do tego zasobu.

**Uaktualnienia.** Python, podobnie jak inne języki programowania, jest nieustannie rozwijany. Dlatego jeśli cokolwiek nie będzie działało zgodnie z oczekiwaniami, zajrzyj do tego zasobu, ponieważ być może instrukcje uległy zmianie.

**Rozwiązania do ćwiczeń.** Warto poświęcić trochę czasu na samodzielne rozwiązywanie zadań zamieszczonych w poszczególnych rozdziałach. Jeżeli napotkałeś trudności i utknąłeś, rozwiązania większości zadań z książki znajdziesz w tym zasobie.

**Ściągi.** Przygotowałem do pobrania pełny zestaw ściąg przedstawiających różne koncepcje związane z programowaniem w Pythonie. Ściągi te znajdziesz w tym zasobie.

# Dlaczego Python?

Każdego roku zastanawiam się, czy kontynuować używanie Pythona, czy przejść do innego języka programowania, prawdopodobnie jednego z tych dopiero pojawiających się w świecie programowania. Jednak wciąż z wielu powodów pozostaję przy Pythonie. Przede wszystkim Python to niezwykle efektywny język — to samo zadanie, a nawet więcej, można wykonać w Pythonie za pomocą mniejszej liczby wierszy kodu niż w innych językach programowania. Ponadto składnia Pythona pomaga w tworzeniu „czystego” kodu. Dzięki temu w porównaniu do innych języków programowania utworzony w ten sposób kod będzie łatwiejszy w odczycie podczas debugowania oraz w trakcie dalszego rozbudowywania programu.

Programiści używają Pythona do wielu celów, między innymi tworzenia gier i aplikacji internetowych, rozwiązywania problemów biznesowych oraz opracowywania wewnętrznych narzędzi wykorzystywanych we wszelkich obszarach zainteresowania firm. Python jest dość często używany w zastosowaniach naukowych, badaniach akademickich oraz w praktycznych rozwiązaniach.

Jednym z najważniejszych powodów, dla których wciąż korzystam z Pythona, jest jego społeczność składająca się z niewiarygodnie różnorodnej i przyjaznej grupy osób. Społeczność jest ważna dla programistów, ponieważ programowanie to najczęściej nie jest praca w pojedynkę. Większość z nas, nawet najbardziej

doświadczeni programiści, musi czasem zapytać o radę innych, którzy już rozwiązywali podobne problemy. Istnienie dobrze zorganizowanej i pomocnej społeczności jest nieocenione, kiedy musisz rozwiązać jakiś problem. Społeczność Pythona jest niezwykle życzliwa dla osób, które podobnie jak Ty dopiero zaczynają poznawać Pythona jako pierwszy język programowania.

Python to doskonały język do nauki programowania, więc rozpoczęijmy już pracę!

# Część I

## Podstawy

CZĘŚĆ PIERWSZA KSIĄŻKI ZAWIERA OMÓWIENIE PODSTAWOWYCH KONCEPCJI PROGRAMOWANIA, KTÓRYCH ZNAJOMOŚĆ JEST NIEZBĘDNA PODCZAS TWORZENIA APLIKACJI W PYTHONIE. WIELE Z TYCH KONCEPCJI ISTNIEJE RÓWNIEŻ W INNYCH JĘZYKACH PROGRAMOWANIA, WIĘC BĘDĄ ONE UŻYTECZNE W TRAKCIE CAŁEJ TWOJEJ KARIERY PROGRAMISTY.

W **rozdziale 1.** zobaczysz, jak zainstalować Pythona na komputerze i utworzyć pierwszy program wyświetlający na ekranie komunikat „Witaj, świecie!”.

W **rozdziale 2.** zobaczysz, jak można przechowywać informacje w zmiennych oraz pracować z wartościami tekstowymi i liczbowymi.

W **rozdziałach 3. i 4.** przejdziemy do list. Lista pozwala przechowywać dowolną ilość informacji w pojedynczej zmiennej, co umożliwia efektywną pracę z danymi. Za pomocą zaledwie kilku wierszy kodu możesz pracować z setkami, tysiącami, a nawet milionami wartości.

W **rozdziale 5.** wykorzystamy konstrukcję `if` do utworzenia kodu, którego sposób działania będzie zależał od wartości warunku: prawda lub fałsz.

W **rozdziale 6.** zobaczysz, jak w Pythonie używać słowników, które pozwalają powiązać ze sobą różne fragmenty informacji. Podobnie jak lista słownik może zawierać dowolną ilość informacji, które chcesz w nim przechowywać.

W **rozdziale 7.** dowiesz się, jak akceptować dane wejściowe od użytkowników i zapewnić interaktywność programu. Ponadto przedstawię pętlę `while`, która wykonuje fragment kodu dopóty, dopóki określony warunek jest prawdziwy.

**W rozdziale 8.** zajmiemy się funkcjami, czyli nazwanymi blokami kodu, które wykonują określone zadania. Funkcję możesz wywołać, gdy zachodzi potrzeba wykonania zdefiniowanego w niej zadania.

**W rozdziale 9.** poznasz klasy umożliwiające modelowanie w kodzie rzeczywistych obiektów, takich jak psy, koty, osoby, samochody, rakiety itd.

**W rozdziale 10.** pokażę, jak pracować z plikami oraz jak obsługiwać błędy, aby działanie programu nie kończyło się nieoczekiwana awarią. Nauczysz się zapisywać dane przed zakończeniem działania programu oraz wczytywać je z powrotem po uruchomieniu programu. Poznasz wyjątki Pythona, które pozwalają przygotować program na wystąpienie ewentualnych błędów, a także umożliwiają mu ich elegancką obsługę.

**W rozdziale 11.** zobaczysz, jak tworzyć testy dla kodu, aby sprawdzić, czy jego działanie jest zgodne z oczekiwaniemi. Dzięki temu będziesz mógł dalej rozbudowywać program bez obaw o wprowadzenie nowych błędów. Testowanie kodu to jedna z pierwszych umiejętności, której nabycie pomaga w przejściu od etapu początkującego programisty do średnio zaawansowanego.

# 1

## Rozpoczęcie pracy



W TYM ROZDZIALE URUCHOMISZ SWÓJ PIERWSZY PROGRAM W PYTHONIE, CZYLI *HELLO\_WORLD.PY*. PRZED EKSPRESOWYM ZAINSTALOWANIEM KODU, SPRAWDZ, CZY NAJNOWSZA WERSJA PYTHONA JEST ZAINSTALOWANA NA KOMPUTERZE.

Jeśli nie, trzeba ją zainstalować. Ponadto potrzebny będzie edytor tekstu przeznaczony do tworzenia kodu źródłowego w Pythonie. Edytor tekstu rozpoznaje kod Pythona i podświetla aktualnie tworzony sekcję, co ułatwia zrozumienie struktury kodu.

## Przygotowanie środowiska programistycznego

Python nieco różni się w poszczególnych systemach operacyjnych, więc musisz wziąć pod uwagę kilka czynników. W kolejnych sekcjach będziesz mieć okazję się upewnić o prawidłowej konfiguracji Pythona w systemie.

### Wersje Pythona

Każdy język programowania ewoluje wraz z pojawianiem się nowych idei i technologii. Nie powinno być więc zaskoczeniem, że twórcy Pythona nieustannie wzbogacają go o nowe możliwości, zapewniające coraz większą wszechstronność temu językowi. Wprawdzie gdy piszę te słowa, najnowszą wersją Pythona jest 3.11, ale wszystkie przykłady przedstawione w książce powinny działać w wydaniu 3.9.

lub nowszym. W tym podrozdziale sprawdzisz, czy Python jest dostępny w używanym przez Ciebie systemie, i dowiesz się, jak go zainstalować, jeśli będziesz potrzebować nowszego wydania języka. Dodatek A zawiera znacznie bardziej szczegółowe omówienie procesu instalacji najnowszej wersji Pythona w najpopularniejszych systemach operacyjnych.

## Wykonanie fragmentu kodu w Pythonie

Python jest dostarczany wraz z interpreterem działającym w powłoce, co pozwala na wypróbowanie kodu bez konieczności zapisywania i uruchamiania całego programu.

W książce będziesz często spotykał fragmenty kodu podobne do przedstawionych poniżej:

---

```
>>> print("Witaj, interpreterze Pythona!")
Witaj, interpreterze Pythona!
```

---

Znaki **>>>**, określane mianem *znaku zachęty Pythona*, wskazują na konieczność użycia powłoki. Tekst wyróżniony pogrubioną czcionką musisz wpisać, a następnie wykonujesz ten fragment kodu, naciskając klawisz *Enter*. Większość przykładów przedstawionych w książce to małe samodzielne programy uruchamiane z poziomu edytora tekstu zamiast powłoki — właśnie w ten sposób będziesz później tworzyć większość kodu. Czasami podstawowe koncepcje będą prezentowane w serii fragmentów kodu wykonywanych w sesji powłoki Pythona, co pozwala na znacznie efektywniejsze zademonstrowanie pewnych zagadnień. Gdy zobaczyś trzy nawiasy ostre w listingu, oznacza to, że masz do czynienia z danymi wyjściowymi sesji w powłoce. Za chwilę przejdziemy do interpretera w używanym przez Ciebie systemie operacyjnym.

Edytor tekstu zostanie użyty do utworzenia prostego programu wyświetlającego komunikat w stylu *Witaj, świecie!*. W świecie programowania od dawna istnieje przekonanie, że wyświetlenie komunikatu „Witaj, świecie!” w pierwszym programie tworzonym podczas nauki nowego języka programowania przyniesie szczęście. Tak prosty program ma również bardzo ważne znaczenie. Jeżeli zostanie prawidłowo wykonany w systemie, to każdy program, który stworzysz w Pythonie, też powinien działać.

## Edytor tekstu VS Code

VS Code to bezpłatny edytor oferujący potężne możliwości i jednocześnie przyjazny początkującym programistom, choć używają go także profesjonalisi. Doskonale sprawdza się podczas tworzenia zarówno prostych, jak i skomplikowanych projektów. Jeżeli przyzwyczaisz się do tego edytora podczas poznawania Pythona, pozostaiesz przy nim, gdy przejdziesz do tworzenia większych i bardziej skomplikowanych

projektów. VS Code można zainstalować w większości nowoczesnych systemów operacyjnych, a także zapewnia obsługę najważniejszych języków programowania, w tym Pythona.

W dodatku B znajdziesz informacje o jeszcze innych dostępnych edytorach tekstu. Jeżeli nie możesz powstrzymać ciekawości, zajrzyj teraz do tego dodatku. Czytelnicy dopiero rozpoczynający przygodę z programowaniem mogą używać VS Code, a nabrawszy doświadczenia, rozważyć przejście do innych edytorów tekstu. W tym rozdziale przedstawię proces instalacji VS Code w różnych systemach operacyjnych.

**UWAGA** *Jeżeli masz już zainstalowany edytor tekstu i wiesz, jak go skonfigurować do uruchamiania programów Pythona, możesz z niego korzystać zamiast z VS Code.*

## Python w różnych systemach operacyjnych

Python to niezależny od platformy język programowania przeznaczony do działania we wszystkich najważniejszych systemach operacyjnych. Każdy utworzony w Pythonie program powinien działać na każdym nowoczesnym komputerze, na którym jest zainstalowany Python. Jednak sama metoda instalacji Pythona zależy od używanego systemu operacyjnego.

W tym podrozdziale dowiesz się, jak zainstalować Pythona i uruchomić w używanym przez Ciebie systemie. Przede wszystkim sprawdź, czy najnowsza wersja Pythona jest dostępna w systemie. Jeżeli jeszcze nie, musisz go zainstalować. W kolejnym kroku zainstalujesz edytor tekstu VS Code. To są jedyne dwa kroki odmienne w poszczególnych systemach operacyjnych.

Następnie utworzysz program Pythona wyświetlający komunikat *Witaj, świecie!* i ewentualnie zajmiesz się usunięciem błędów, jeśli program nie będzie działać. Ten proces omówię tutaj dla każdego najważniejszego systemu operacyjnego, dzięki temu otrzymasz środowisko programowania przyjazne początkującym programistom.

### Python w systemie Windows

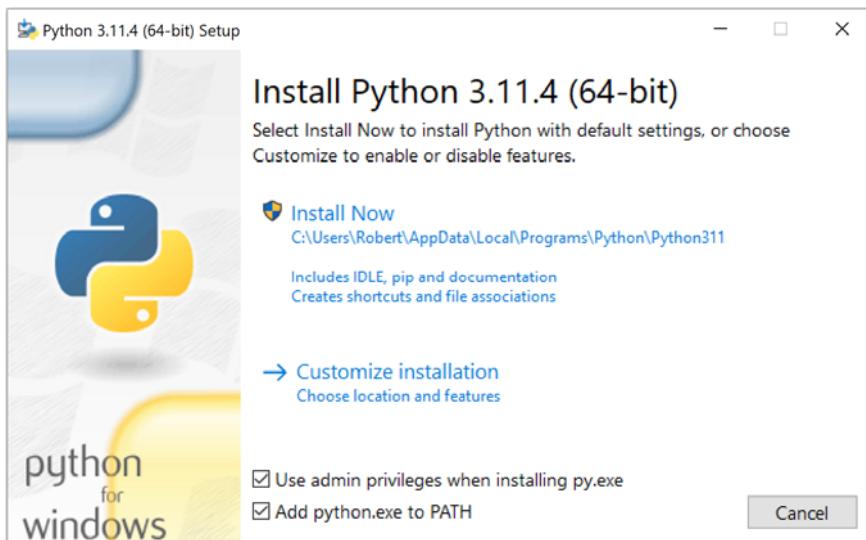
System Windows nie zawsze jest dostarczany z Pythonem, więc prawdopodobnie będziesz musiał go pobrać i zainstalować, a następnie pobrać i zainstalować również VS Code.

#### Instalacja Pythona

Przede wszystkim sprawdź, czy Python jest zainstalowany w systemie. Przejdz do wiersza poleceń, wpisując **cmd** w menu *Start*, lub z wciśniętym klawiszem *Shift* kliknij prawym przyciskiem myszy na pulpicie i wybierz z menu kontekstowego opcję *Otwórz tutaj okno programu PowerShell*. W oknie wiersza poleceń wpisz małymi literami **python**. Jeżeli zobaczysz znak zachęty Pythona (czyli **>>>**), oznacza to, że Python jest już zainstalowany w systemie. Natomiast jeśli otrzymasz komunikat

błędu informujący o nieroznianym poleceniu `python` lub zostanie uruchomiona aplikacja Microsoft Store, oznacza to brak Pythona w systemie. W takim przypadku zamknij tę aplikację — znacznie lepszym rozwiązaniem jest pobranie oficjalnego pakietu instalacyjnego niż użycie wersji dostarczanej przez Microsoft.

Jeżeli Python nie jest zainstalowany lub jeśli masz wersję języka wcześniejszą niż 3.9, musisz pobrać instalator Pythona dla Windows. Przejdz na stronę <https://www.python.org/downloads/>. Powinieneś zobaczyć przycisk pozwalający na pobranie Pythona w najnowszej dostępnej wersji 3. Kliknij go, co powinno automatycznie rozpoczęć pobieranie odpowiedniego instalatora dla używanego przez Ciebie systemu. Po pobraniu pliku uruchom instalator. Upewnij się, że została wybrana opcja `Add python.exe to PATH`, co niezwykle ułatwi prawidłową konfigurację systemu. Na rysunku 1.1 pokazałem tę opcję zaznaczoną w instalatorze.



Rysunek 1.1. Upewnij się, że zostało zaznaczone pole wyboru `Add python.exe to PATH`

## Uruchomienie Pythona w sesji wiersza poleceń

Otwórz okno wiersza poleceń i wpisz małymi literami `python`. Jeżeli zobaczyς znak zachęty Pythona (czyli `>>>`), oznacza to, że Windows znalazł zainstalowaną przed chwilą wersję Pythona:

```
C:\> python
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit
↪(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

**UWAGA** Jeżeli otrzymałeś inne dane wyjściowe, zajrzyj do dodatku A, w którym znajdziesz znacznie dokładniejsze informacje dotyczące instalacji Pythona.

Wprowadź poniższy wiersz kodu w sesji Pythona w wierszu polecenia i upewnij się, że otrzymałeś dane wyjściowe w postaci komunikatu **Witaj, świecie Pythona!**:

---

```
>>> print("Witaj, świecie Pythona!")
Witaj, świecie Pythona!
>>>
```

---

W wierszu polecenia powinien zostać wyświetlony komunikat **Witaj, świecie Pythona!**. Za każdym razem, gdy chcesz wykonać fragment kodu Pythona, przejdź do wiersza polecenia, a następnie uruchom powłokę Pythona. Aby zakończyć sesję powłoki Pythona, naciśnij klawisze **Ctrl+Z** i później **Enter** lub wydaj polecenie **exit()**.

## Instalacja VS Code

Program instalacyjny VS Code znajdziesz w witrynie <https://code.visualstudio.com>. Kliknij łącze **Download**, co spowoduje pobranie odpowiedniego pliku i uruchom program instalacyjny. Następnie pomiń kolejne punkty i przejdź od razu do podrozdziału „Uruchomienie programu typu „Witaj, świecie!””.

## Python w systemie macOS

W najnowszych wydaniach systemu macOS nie ma zainstalowanego domyślnie Pythona. W tej sekcji dowiesz się, jak zainstalować najnowszą wersję Pythona. Ponadto zainstalujesz edytor VS Code i sprawdzisz, czy jest on prawidłowo skonfigurowany.

**UWAGA** W starszych wersjach systemu macOS było zainstalowane starsze wydanie Pythona, wersja 2. Jest ona uznawana za przestarzałą i nie należy jej używać.

## Sprawdzenie, czy Python 3 jest zainstalowany

Przejdz do okna programu Terminal (znajdziesz go w katalogu *Aplikacje/Narzędzia/Terminal*). Ewentualnie naciśnij klawisze **Command + Spacja**, wpisz **terminal** i wciśnij **Enter**. Aby sprawdzić, czy Python jest już zainstalowany w systemie, wydaj polecenie **python3**, przy czym napisz je małymi literami. Prawdopodobnie otrzymasz komunikat dotyczący instalacji *narzędzi programistycznych działających w powłoce*. Znacznie lepiej jest je zainstalować już po zainstalowaniu Pythona. Dlatego jeśli zobaczyś ten komunikat, zamknij okno dialogowe, w którym został wyświetlony.

Jeżeli wyświetcone dane wyjściowe wskazują na wersję 3.9 lub nowszą, możesz od razu przejść do punktu „Użycie Pythona w sesji powłoki”. Natomiast

w przypadku posiadania w systemie wcześniejszej niż 3.9 wersji Pythona informacje zamieszczone w następnym podpunkcie wykorzystaj do zainstalowania najnowszej wersji Pythona.

Pamiętaj, aby każde pojawiające się w książce polecenie `python` zastępować w systemie macOS poleceniem `python3`. W ten sposób będziesz używać Pythona 3. W większości systemów macOS polecenie `python` prowadzi do przestarzałej wersji Pythona, która powinna być używana jedynie przez wewnętrzne narzędzia systemowe. Ewentualnie to polecenie do niczego nie prowadzi i spowoduje wygenerowanie komunikatu błędu.

## Instalowanie najnowszej wersji Pythona

Przejdź na stronę <https://www.python.org/downloads/>. Powinieneś zobaczyć przycisk umożliwiający pobranie Pythona w najnowszej dostępnej wersji 3. Kliknij go, co powinno automatycznie rozpoczęć pobieranie odpowiedniego pliku dla używanego przez Ciebie systemu. Po pobraniu pliku uruchom instalator.

Po zakończeniu instalacji powinno pojawić się okno Finder'a. Dwukrotnie kliknij plik `Install Certificates.command`. Dzięki niemu będzie można znacznie łatwiej instalować biblioteki dodatkowe, niezbędne w pracy nad rzeczywistymi projektami, w tym nad projektami omówionymi w części drugiej książki.

## Użycie Pythona w sesji powłoki

Fragmenty kodu Pythona możesz uruchamiać, przechodząc do aplikacji Terminal, a następnie wydając polecenie `python3`.

---

```
$ python3
Python 3.x.x (v3.11.0:eb0004c271, Jun . . . , 10:03:01)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

---

To polecenie spowoduje uruchomienie sesji powłoki Pythona. Zobaczysz znak zachęty Pythona, `>>>`, wskazujący, że system macOS znalazł zainstalowaną przed chwilą wersję Pythona.

Wydaj w powłoce następujące polecenie:

---

```
>>> print("Witaj, interpreterze Pythona!")
Witaj, interpreterze Pythona!
>>>
```

---

Powinieneś zobaczyć komunikat wyświetlony bezpośrednio w bieżącym oknie aplikacji Terminal. Pamiętaj o możliwości zakończenia pracy z powłoką Pythona przez naciśnięcie klawiszy `Ctrl+D` lub wydanie polecenia `exit()`.

**UWAGA** W nowszych wersjach systemu macOS jako znak zachęty powłoki zobaczyesz znak procentu, %, a nie znak dolara, \$.

## Instalacja VS Code

Program instalacyjny VS Code znajdziesz w witrynie <https://code.visualstudio.com>. Kliknij łącze *Download*, co spowoduje pobranie odpowiedniego pliku. Następnie rozpakuj archiwum i przeciągnij ikonę *Visual Studio Code* na katalog *Aplikacje*.

Pomiń kolejne punkty dotyczące instalacji Pythona w Linuksie i przejdź od razu do podrozdziału „Uruchomienie programu typu „Witaj, świecie!””.

## Python w systemach z rodziny Linux

Systemy z rodziny Linux zostały zaprojektowane do programowania, więc Python jest już zainstalowany praktycznie w każdym z nich. Od osób tworzących i obsługujących systemy Linux oczekuje się umiejętności programowania i zachęca je do tego. Dlatego też aby zacząć programować, nie trzeba wiele robić — wystarczy wprowadzić kilka zmian w ustawieniach i można zacząć programować.

## Sprawdzenie zainstalowanej wersji Pythona

Przejdź do powłoki lub otwórz okno terminala w systemie (w Ubuntu możesz nacisnąć klawisze *Ctrl+Alt+T*). Aby sprawdzić zainstalowaną wersję Pythona, napisz małymi literami polecenie **python3**. Jeżeli Python jest zainstalowany, to polecenie spowoduje uruchomienie interpretera Pythona. Powinieneś zobaczyć informacje o zainstalowanej wersji Pythona oraz znak zachęty **>>>**, po którym możesz zacząć wprowadzać inne polecenia Pythona, na przykład takie, jakie pokazalem poniżej:

---

```
$ python3
Python 3.10.4 (main, Apr . . . , 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

---

Powyższe dane wyjściowe potwierdzają, że Python 3.10.4 to aktualnie do-myślna wersja tego języka w systemie. Gdy już uzyskasz te informacje, naciśnij klawisze *Ctrl+D* lub wydaj polecenie **exit()**, aby opuścić powłokę Pythona i powrócić do standardowej powłoki systemu. Pamiętaj tylko, aby każde pojawiające się w książce polecenie **python** zastępować poleceniem **python3**.

Aby uruchamiać przykłady przedstawione w książce, musisz mieć Pythona w wersji 3.9 lub nowszej. Jeśli z jakiegokolwiek powodu używana przez Ciebie dystrybucja ma zainstalowanego Pythona w starszej wersji niż 3.9 lub jeśli chcesz uaktualnić obecną wersję do najnowszej dostępnej, zajrzyj do dodatku A.

## Wykonanie kodu Pythona w powłoce

Fragmenty kodu Pythona możesz spróbować wykonać z poziomu powłoki, co wymaga użycia wymienionego wcześniej polecenia `python` lub `python3`, z którego już skorzystałeś podczas sprawdzania zainstalowanej wersji języka. Ponownie wydaj to samo polecenie, ale tym razem w powłoce Pythona wprowadź następujący wiersz kodu:

---

```
>>> print("Witaj, interpreterze Pythona!")
Witaj, interpreterze Pythona!
>>>
```

---

Powinieneś zobaczyć komunikat wyświetlany bezpośrednio w powłoce. Pamiętaj o możliwości zakończenia pracy z interpreterem Pythona przez naciśnięcie klawiszy `Ctrl+D`, lub też wywołanie funkcji `exit()`.

## Instalacja VS Code

W systemie Ubuntu edytor tekstu VS Code znajdziesz bezpłatnie w oficjalnym sklepie z oprogramowaniem Ubuntu. Kliknij ikonę Oprogramowanie Ubuntu, wpisz w polu wyszukiwania `vscode`. Kliknij aplikację *Visual Studio Code* (czasami nazywaną `vscode`), a następnie kliknij przycisk *Zainstaluj*. Po zainstalowaniu edytora poszukaj go w systemie i uruchom.

# Uruchomienie programu typu „Witaj, świecie!”

Mając zainstalowane najnowsze wersje Pythona i VS Code, jesteś prawie gotowy do uruchomienia pierwszego programu napisanego w Pythonie z użyciem edytora tekstu. Jednak zanim to nastąpi, musisz przeprowadzić konfigurację edytora, aby korzystał z odpowiedniej wersji Pythona w systemie.

## Instalacja w VS Code rozszerzenia przeznaczonego do obsługi Pythona

Edytor VS Code działa z wieloma różnymi językami programowania. Aby w pełni wykorzystać jego możliwości podczas pracy z Pythonem, konieczne jest zainstalowanie rozszerzenia o nazwie *Python*. Pomaga ono w tworzeniu, edytowaniu i uruchamianiu programów Pythona.

Aby zainstalować to rozszerzenie, kliknij ikonę *Manage* w lewym dolnym rogu okna aplikacji VS Code. W wyświetlonym menu kliknij *Extensions*. W polu wyszukiwania wpisz `python` i kliknij element przedstawiający rozszerzenie o nazwie *Python*. (Jeżeli widzisz więcej niż jedno rozszerzenie o nazwie *Python*, wybierz to od Microsoftu). Kliknij *Install* i zainstaluj wszelkie inne narzędzia

dodatkowe wymagane przez system do ukończenia instalacji. Jeżeli otrzymasz komunikat informujący o konieczności instalacji Pythona, mimo że masz go za-instalowanego w systemie, możesz spokojnie zignorować ten komunikat.

**UWAGA** Jeżeli używasz systemu macOS i zobaczysz okno dialogowe informujące o konieczności zainstalowania narzędzi programistycznych, kliknij przycisk Zainstaluj. Pojawi się komunikat informujący o wyjątkowo długim czasie instalacji. Nie przerazaj się, ponieważ w przypadku względnie szybkiego połączenia z internetem instalacja zakończy się w ciągu 10 – 20 minut.

## Uruchomienie programu typu „Witaj, świecie!”

Aby przygotować pierwszy program, w dogodnej lokalizacji utwórz katalog dla projektów i nadaj mu nazwę, na przykład *projekty\_pythona*. W nazwach katalogów i plików najlepiej jest używać jedynie małych liter i znaków podkreślenia zamiast spacji, co wynika ze stosowanej przez Pythona konwencji nazewnictwa. Wprawdzie można go utworzyć w innym miejscu niż na pulpicie, ale praca z przykładami zamieszczonymi w książce będzie łatwiejsza, gdy katalog *projekty\_pythona* umieścis na pulpicie.

Powróć do edytora VS Code i zamknij okno *Get Started*, jeśli wciąż jest wyświetlane na ekranie. Utwórz nowy plik za pomocą opcji menu *File/New File* lub naciskając klawisze *Ctrl+N* (*Command+N* w systemie macOS). Zapisz plik o nazwie *hello\_world.py* w utworzonym wcześniej katalogu (*projekty\_pythona*). Rozszerzenie *.py* wskazuje edytoriowi VS Code, że dany plik będzie zawierał kod Pythona. Ponadto informuje go o sposobie uruchamiania programu i kolorowania kodu, co znacznie ułatwia pracę programistom.

Po zapisaniu pliku umieść w nim poniższy wiersz kodu:

*Plik hello\_world.py:*

---

```
print("Witaj, świecie Pythona!")
```

---

Aby uruchomić ten program, wybierz opcję menu *Run/Run Without Debugging* lub naciśnij klawisze *Ctrl+F5*. W dolnej części okna edytora VS Code pojawi się ekran terminala wraz z następującymi danymi wyjściowymi:

---

```
Witaj, świecie Pythona!
```

---

Prawdopodobnie zostaną wyświetlone dodatkowe dane wyjściowe informujące o interpreterze Pythona użytym do wykonania tego programu. Jeżeli chcesz uprościć wyświetlane informacje do postaci zaledwie danych wyjściowych programu, zajrzyj do dodatku B. Znajdziesz w nim również przydatne podpowiedzi, jak znacznie efektywniej korzystać z edytora VS Code.

Jeżeli nie otrzymałeś powyższych danych wyjściowych, dokładnie sprawdź każdy znak we wprowadzonym wierszu kodu. Czy przypadkiem nie użyłeś dużej litery w poleceniu `print`? Czy na pewno ująłeś komunikat w cudzysłów lub apostrof? Języki programowania oczekują użycia konkretnej składni i jeśli nie dostosujesz się do tych wymagań, zostaną wygenerowane komunikaty błędów. Jeżeli nie możesz sobie poradzić z uruchomieniem programu, zajrzyj do podrozdziału „Rozwiązywanie problemów podczas instalacji” w dalszej części tego rozdziału.

## Rozwiązywanie problemów podczas instalacji

Jeżeli nie możesz uruchomić programu `hello_world.py`, poniżej przedstawiam kilka podpowiedzi, jak rozwiązać ten problem:

- Jeżeli program zawiera poważny błąd, wówczas Python wyświetla **stos wywołań** (*traceback*). W takim przypadku Python analizuje plik i próbuje zgłosić problem. Wyświetlony stos wywołań może być wskazówką, jaki problem uniemożliwił uruchomienie programu.
- Odejdź od komputera, zrób sobie krótką przerwę, a następnie wróć i spróbuj ponownie. Pamiętaj, że składnia ma duże znaczenie w programowaniu, więc nawet brakujący dwukropki, cudzysłów, apostrof lub błędnie dopasowany nawias może uniemożliwić uruchomienie programu. Ponownie przeczytaj odpowiednie fragmenty tego rozdziału, przejrzyj to, co dotychczas zrobileś, i spróbuj znaleźć, gdzie popełniłeś błąd.
- Zaczni od początku. Prawdopodobnie nie będziesz musiał niczego odinstalowywać z systemu, ale rozsądnie będzie, jeśli usuniesz plik `hello_world.py` i ponownie go utworzysz.
- Poproś kogoś innego o wykonanie przedstawionej w tym rozdziale procedury na Twoim lub na innym komputerze i dokładnie obserwuj podejmowane przez niego działania. Być może pominąłeś jeden mały krok, który zostanie wykonany przez inną osobę.
- Zapoznaj się z procedurą instalacji zamieszczoną w dodatku A. Pewne szczegóły przedstawione w tym dodatku mogą pomóc w rozwiązyaniu problemu.
- Poszukaj jakiegoś programisty Pythona i poproś o pomoc w skonfigurowaniu systemu. Jeżeli dobrze się rozejrzesz wokół siebie, prawdopodobnie znajdziesz osobę zajmującą się programowaniem w Pythonie.
- Opisana w tym rozdziale procedura konfiguracji środowiska jest dostępna również w internecie na stronie [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Być może polecenia przedstawione w wersji online sprawdzą się lepiej w Twoim przypadku, ponieważ możesz je po prostu skopiować i wkleić, a także kliknąć łącza do potrzebnych Ci zasobów.

- Poproś o pomoc w internecie. W dodatku C wymieniłem wiele źródeł internetowych, między innymi fora i czaty, na których możesz poprosić o pomoc osoby zajmującą się programowaniem w Pythonie. Być może ktoś inny spotkał się już z problemem takim jak Twój.

Nie obawiaj się prosić o pomoc bardziej doświadczonych programistów. Każdy z nich musiał się kiedyś zmierzyć z trudnym do rozwiązania problemem i większość z nich chętnie pomoże Ci w prawidłowej konfiguracji systemu. Jeżeli tylko potrafisz jasno i czytelnie określić to, co próbujesz osiągnąć, jakie kroki podjęłeś i jakie otrzymałeś wyniki, masz duże szanse, że znajdziesz się ktoś, kto będzie w stanie skutecznie Ci pomóc. Jak wspomniałem we wprowadzeniu, społeczność Pythona jest niezwykle przyjazna dla poczatkujących programistów.

Python powinien działać doskonale na w miarę nowoczesnym komputerze. Problemy pojawiające się na początku mogą być frustrujące, ale naprawdę warto spróbować je rozwiązać. Gdy tylko uda Ci się uruchomić program *hello\_world.py*, możesz przystąpić do nauki Pythona, a Twoja praca programisty stanie się bardziej interesująca i satysfakcjonująca.

## Uruchamianie programów Pythona z poziomu powłoki

Większość tworzonych przez nas programów w edytorze tekstu będziesz uruchamiać bezpośrednio z poziomu edytora. Jednak czasami użyteczne będzie uruchomienie programu z poziomu powłoki systemu. Na przykład chcesz uruchomić istniejący program bez otwierania go do edycji.

Masz taką możliwość w każdym systemie, w którym jest zainstalowany Python, o ile wiesz, jak uzyskać dostęp do katalogu z plikiem programu. Aby wypróbować przedstawione poniżej sposoby, upewnij się, że plik *hello\_world.py* zapisalesz w katalogu *projekty\_pythona* umieszczonym na pulpicie.

### W systemie Windows

Polecenie `cd` (skrót od *change directory*, czyli zmień katalog) jest używane do poruszania się po systemie plików z poziomu wiersza polecenia. Z kolei polecenie `dir` (skrót od *directory*, czyli katalog) wyświetla wszystkie pliki znajdujące się w bieżącym katalogu.

Przejdź do wiersza polecenia, a następnie wydaj poniższe polecenia, aby uruchomić program, którego kod źródłowy znajduje się w pliku *hello\_world.py*:

---

```
C:\> cd Projekty\projekty_pythona
C:\Projekty\projekty_pythona> dir
hello_world.py
C:\Projekty\projekty_pythona> python hello_world.py
Witaj, świecie Pythona!
```

---

Najpierw użyliśmy polecenia `cd` w celu przejścia do katalogu `projekty_pythona` znajdującego się w katalogu `Pulpit`. Następnie za pomocą polecenia `dir` sprawdziliśmy, czy ten katalog na pewno zawiera plik `hello_world.py`. Ostatnim krokiem było uruchomienie programu przez wydanie polecenia `python hello_world.py`.

Większość programów działa doskonale, kiedy uruchomimy je z poziomu edytora, jednak gdy staną się znacznie bardziej skomplikowane, może się zdarzyć, że będziemy musieli uruchomić je bezpośrednio z poziomu powłoki systemu.

## W systemach macOS i Linux

Uruchomianie programu Pythona z poziomu powłoki systemu odbywa się w systemach Linux i macOS w dokładnie taki sam sposób. Za pomocą polecenia `cd` (skrót od *change directory*, czyli zmień katalog) możesz poruszać się po systemie plików z poziomu powłoki. Z kolei polecenie `ls` (skrót od *list*, czyli wyświetl) wyświetla listę wszystkich nieukrytych plików znajdujących się w bieżącym katalogu.

Przejdź do powłoki, a następnie wydaj poniższe polecenia, aby uruchomić program `hello_world.py`:

```
-$ cd Pulpit/projekty_pythona/  
~/Pulpit/projekty_pythona$ ls  
hello_world.py  
~/Pulpit/projekty_pythona$ python3 hello_world.py  
Witaj, świecie Pythona!
```

Najpierw użyliśmy polecenia `cd` w celu przejścia do katalogu `projekty_pythona` znajdującego się w katalogu `Pulpit`. Następnie za pomocą polecenia `ls` sprawdziliśmy, czy ten katalog na pewno zawiera plik `hello_world.py`. Ostatnim krokiem było uruchomienie programu przez wydanie polecenia `python3 hello_world.py`.

Większość programów działa doskonale po ich uruchomieniu z poziomu edytora tekstu. Gdy jednak staną się znacznie bardziej skomplikowane, wówczas może się zdarzyć, że trzeba będzie je uruchamiać bezpośrednio z poziomu powłoki systemu.

### ZRÓB TO SAM

Ćwiczenia w tym rozdziale mają charakter poznawczy. Jednak począwszy od rozdziału 2., do rozwiązywania ćwiczeń będziesz musiał wykorzystywać zdobytą wcześniej wiedzę.

**1.1. Witryna `python.org`.** Zapoznaj się ze stroną domową Pythona (<http://python.org/>) i spróbuj wyszukać interesujące Cię tematy. Gdy zdobędziesz nieco więcej doświadczenia w programowaniu w Pythonie, poszczególne części tej witryny staną się dla Ciebie znacznie użyteczniejsze.

**1.2. Błędy w programie typu „Witaj, świecie!”.** Otwórz utworzony w tym rozdziale plik *hello\_world.py*. Wprowadź dowolny błąd w wierszu kodu, a następnie uruchom program. Czy potrafisz wprowadzić zmianę powodującą wygenerowanie komunikatu błędu? Czy ten komunikat błędu ma dla Ciebie sens? Czy potrafisz wprowadzić zmianę niepowodującą wygenerowania błędu? Na jakiej podstawie uważasz, że ta zmiana nie jest błędem?

**1.3. Niczym nieograniczone umiejętności.** Gdybyś posiadał nieograniczone umiejętności w zakresie programowania, jaki program byś stworzył? Dzięki tej książce masz nauczyć się programować. Jeżeli będziesz miał na uwadze pewien cel, który sam sobie wyznaczysz, zyskasz możliwość częściowego wykorzystania nowo nabitych umiejętności. Teraz jest więc doskonały moment na określenie tego, co chciałbyś stworzyć. Dobrym rozwiązaniem jest prowadzenie dziennika „idei”, do którego będziesz mógł się odwoływać, rozpoczynając pracę nad nowym projektem. Poświęć kilka minut na opisanie trzech programów, które chciałbyś stworzyć.

## Podsumowanie

W tym rozdziale dowiedziałeś się co nieco na temat Pythona i zainstalowałeś go w systemie. Ponadto zainstalowałeś edytor tekstu, aby ułatwić sobie pracę podczas tworzenia kodu Pythona. Uruchamiałeś fragmenty kodu Pythona w powłoce oraz utworzyłeś pierwszy rzeczywisty program w pliku *hello\_world.py*. Prawdopodobnie dowiedziałeś się także co nieco na temat usuwania błędów.

W następnym rozdziale poznasz różne rodzaje danych, z jakimi można pracować w programach wykorzystujących Pythona. Ponadto zobaczysz, jak używać zmiennych.

# 2

## Zmienne i proste typy danych



W TYM ROZDZIALE POZNASZ RÓŻNE RODZAJE DANYCH, Z JAKIMI MOŻNA PRACOWAĆ W PYTHONIE. PONADTO DOWIESZ SIĘ, JAK PRZECHOWYWAĆ DANE W ZMIENNYCH ORAZ JAK KORZYSTAĆ Z TYCH ZMIENNYCH W PROGRAMACH.

### Co tak naprawdę dzieje się po uruchomieniu `hello_world.py`?

Przekonajmy się, co tak naprawdę się dzieje, gdy próbujesz uruchomić program `hello_world.py` w Pythonie. Jak się okazuje, Python wykonuje dość dużą ilość pracy, pomimo uruchamiania niezwykle prostego programu.

*Plik `hello_world.py`:*

---

```
print("Witaj, świecie Pythona!")
```

---

Po uruchomieniu powyższego programu powinieneś otrzymać następujące dane wyjściowe:

---

```
Witaj, świecie Pythona!
```

---

Kiedy uruchamiasz plik *hello\_world.py*, rozszerzenie *.py* wskazuje na zdefiniowanie w nim kodu źródłowego Pythona. Edytor tekstu najpierw przetwarza ten plik za pomocą *interpretera Pythona*, który odczytuje go wiersz po wierszu i określa znaczenie każdego znajdującego się w nim słowa. Na przykład gdy interpreter natrafi na słowo *print*, wyświetla na ekranie to wszystko, co zostało ujęte w cudzysłów znajdujący się w nawiasie.

Gdy piszesz program, edytor na różne sposoby koloruje poszczególne fragmenty kodu źródłowego tego programu. Rozpoznaje, że słowo *print* oznacza nazwę funkcji, więc wyświetla je w pewnym kolorze. Ustala też, że ciąg tekstowy "Witaj, świecie Pythona!" nie jest kodem Pythona i dlatego wyświetla go w innym kolorze. Ta funkcja edytora nosi nazwę *kolorowania składni*. Uznasz ją za niezwykle pomocną, gdy zaczniesz tworzyć własne programy.

## Zmienne

Spróbujmy teraz użyć zmiennej w programie *hello\_world.py*. Na początku pliku wstaw nowy wiersz, a następnie zmodyfikuj drugi, aby kod przyjął taką postać:

*Plik hello\_world.py:*

---

```
message = "Witaj, świecie Pythona!"  
print(message)
```

---

Uruchom ten program i zobacz, co się stanie. Powinieneś otrzymać dokładnie te same dane wyjściowe, które zostały wygenerowane już wcześniej:

---

```
Witaj, świecie Pythona!
```

---

W kodzie umieściliśmy *zmienną* o nazwie *message*. Każda zmienna przechowuje *wartość*, czyli informacje powiązane z tą zmienną. W omawianym przykładzie wartością jest komunikat „Witaj, świecie Pythona!”.

Dodanie zmiennej w kodzie oznacza nieco więcej pracy dla interpretera Pythona. Podczas przetwarzania pierwszego wiersza powiązuje on komunikat „Witaj, świecie Pythona!” ze zmienną *message*. Następnie po przejściu do drugiego wiersza wyświetla na ekranie wartość przypisaną wcześniej zmiennej *message*.

Rozbudujemy teraz ten program, modyfikując plik *hello\_world.py* i dodając do niego polecenia przeznaczone do wyświetlenia drugiego komunikatu. Wstaw więc pusty wiersz po dotychczasowym kodzie, a następnie dodaj dwa nowe.

---

```
message = "Witaj, świecie Pythona!"  
print(message)  
  
message = 'Witaj, świecie książki "Python. Instrukcje dla programisty"!'  
print(message)
```

---

Teraz po uruchomieniu programu *hello\_world.py* powinieneś otrzymać dwa wiersze danych wyjściowych:

---

Witaj, świecie Pythona!  
Witaj, świecie książki "Python. Instrukcje dla programisty"!

---

Wartość zmiennej w programie można zmienić w dowolnym momencie, a Python zawsze będzie monitorować jej bieżącą wartość.

## Nadawanie nazw zmiennym i używanie zmiennych

Używając zmiennych w Pythonie, trzeba stosować się do kilku reguł i zaleceń. Złamanie niektórych reguł spowoduje powstanie błędów, natomiast inne zalecenia mają pomóc w tworzeniu kodu, który będzie łatwiejszy do odczytania i zrozumienia. Postaraj się pamiętać o wymienionych poniżej regułach dotyczących nazw zmiennych:

- Nazwy zmiennych mogą zawierać jedynie litery, cyfry i znaki podkreślenia. Nazwa może rozpoczynać się od litery lub znaku podkreślenia, ale nie od cyfry. Na przykład zmiennej można nadać nazwę `message_1`, ale już nie `1_message`.
- Niedozwolone jest stosowanie spacji w nazwach zmiennych, choć znak podkreślenia można stosować w charakterze separatora słów. Na przykład `greeting_message` to prawidłowa nazwa zmiennej, podczas gdy `greeting message` spowoduje wygenerowanie błędu.
- Unikaj użycia w nazwach zmiennych słów kluczowych Pythona oraz nazw funkcji. Oznacza to, że nie powinieneś używać słów, które w Pythonie są zarezerwowane do określonych celów programistycznych. Przykładem słowa zarezerwowanego jest `print`. (Więcej informacji na ten temat znajdziesz w dodatku A).
- Nazwa zmiennej powinna być krótka, choć jednocześnie czytelna. Na przykład nazwa `name` jest lepsza niż `n`, `student_name` lepsza niż `s_n`, a `name_length` lepsza niż `length_of_persons_name`.
- Ostrożnie używaj małej litery *l* i wielkiej litery *O*, ponieważ łatwo je pomylić z cyframi — odpowiednio *1* i *0*.

Może upływać trochę czasu, zanim nabędziesz wprawy w wybieraniu dobrych nazw dla zmiennych, co będzie szczególnie istotne wtedy, kiedy tworzone programy staną się bardziej interesujące i skomplikowane. Gdy zaczniesz tworzyć więcej własnych programów i analizować kod opracowany przez innych, jeszcze lepiej zrozumiesz, co oznacza stosowanie jasnych i czytelnych nazw.

**UWAGA** *Zmienne, których teraz używamy, powinny mieć nazwy zapisywane małymi literami. Wprawdzie nie otrzymasz komunikatu błędu, jeśli użyjesz wielkich liter, ale zmienne zapisane wielkimi literami mają znaczenie specjalne, co wyjaśnię w dalszej części rozdziału.*

# Unikanie błędów związanych z nazwami podczas używania zmiennych

Każdy programista popełnia błędy, a większość popełnia je każdego dnia. Wprawdzie dobry programista również może popełniać błędy, ale jednocześnie wie, jak efektywnie na nie reagować. Przeanalizujmy teraz błąd, który prawdopodobnie wcześniej czy później popełnisz, i zobaczymy, jak możesz go usunąć.

Utworzymy teraz fragmentu kodu celowo generujący komunikat błędu. Wprowadź dwa poniższe polecenia, w tym celowo błędnie zapisane słowo *mesage*, które w kodzie zostało pogrubione:

```
message = 'Witaj, Czytelniku książki "Python. Instrukcje dla programisty"!'  
print(mesage)
```

Gdy w programie wystąpi błąd, interpreter Pythona stara się jak najlepiej pomóc w ustaleniu przyczyny problemu. Dlatego też interpreter wyświetla stos wywołań, jeśli uruchomienie programu zakończyło się niepowodzeniem. Ten *stos wywołań* jest zapisem działań, które spowodowały pojawienie się problemu podczas wykonywania kodu. Poniżej przedstawiłem przykład stosu wywołań, który zostanie wyświetlony po przypadkowym błędny podaniu nazwy zmiennej:

```
Traceback (most recent call last):  
  File "hello_world.py", line 2, in <module> ①  
    print(mesage) ②  
          ^~~~~~  
NameError: name 'mesage' is not defined. Did you mean: 'message'? ③
```

W wierszu **①** danych wyjściowych mamy informacje o błędzie, który wystąpił w wierszu 2. pliku kodu źródłowego *hello\_world.py*. Interpreter wyświetla ten numer wiersza, aby pomóc w szybszym wychwyceniu błędu (patrz wiersz **②**) i podaje rodzaj znalezionej błędu (patrz wiersz **③**). W omawianym przypadku mamy *błąd nazwy* (typ *NameError*), a wyświetlony komunikat informuje, że zmienna *mesage* nie została zdefiniowana. Python nie potrafi zidentyfikować zmiennej o podanej nazwie. Tego rodzaju błąd zwykle wskazuje, że zapomniałeś przypisać wartość zmiennej przed jej użyciem lub pomyliłeś się podczas wpisywania nazwy zmiennej. Jeżeli Python znajdzie zmienną o nazwie podobnej do nieroznianej, wówczas zapyta, czy to właśnie tej znalezionej przez niego nazwy chcesz użyć w kodzie.

Oczywiście w omawianym przykładzie mamy do czynienia z drugą z wymienionych sytuacji: zapomnieliśmy o literze *s* w nazwie zmiennej *message* użytej w wierszu drugim. Interpreter Pythona nie sprawdza kodu pod kątem poprawności pisowni słów, ale może ustalić, czy nazwy zmiennych są używane spójnie. Zobacz na przykład, co się stanie, gdy w innym miejscu kodu również użyjemy nieprawidłowo zapisanej nazwy zmiennej *message*:

---

```
mesage = 'Witaj, Czytelniku książki "Python. Instrukcje dla programisty"!'  
print(mesage)
```

---

W takim przypadku uruchomienie programu zakończy się powodzeniem:

---

```
Witaj, Czytelniku książki "Python. Instrukcje dla programisty"!
```

---

Nazwy zmiennych zostały użyte spójnie, więc Python nie ma problemu z wykonaniem tego kodu. Komputery ściśle stosują się do wydawanych poleceń, choć nie zwracają uwagi na poprawną pisownię słów. Dlatego też podczas tworzenia nazw zmiennych i pisania kodu nie musisz brać pod uwagę reguł pisowni i gramatyki.

Większość tego rodzaju błędów to drobne pomyłki — zwykle dotyczą pojęcia dynczego znaku w jednym wierszu kodu źródłowego programu. Jeżeli poświęcasz dużo czasu na wyszukiwanie jednego z tego typu błędów, to powinieneś wiezieć, że nie jesteś sam. Wielu doświadczonych i utalentowanych programistów marnuje godziny na wychwytywanie tego rodzaju drobnych błędów. Nie przejmuj się tym i przejdź dalej, ponieważ z takimi sytuacjami będziesz się często spotykał w swojej karierze programisty.

## Zmienna to etykieta

Zmienna jest bardzo często przedstawiana jako pojemnik przeznaczony do przechowywania wartości. Wprawdzie takie porównanie może okazać się pomocne, gdy dopiero zaczynasz poznawać zmienne, ale nie odzwierciedla prawidłowo sposobu, w jaki są one wewnętrznie przedstawiane w Pythonie. Znacznie lepiej jest traktować zmienną jako etykietę, którą można przypisać wartości. Można również stwierdzić, że zmienna odwołuje się do określonej wartości.

Takie rozróżnienie prawdopodobnie nie będzie miało większego znaczenia w pierwszych tworzonych programach, ale mimo to warto się o tym dowiedzieć wcześniej niż później. W pewnym momencie zetkniesz się z nieoczekiwanyem zachowaniem zmiennej i wówczas znajomość sposobu jej działania pomoże w ustaleniu, co tak naprawdę dzieje się w kodzie.

**UWAGA** *Najlepszym sposobem na poznanie nowych koncepcji w programowaniu jest używanie ich we własnych programach. Jeżeli utkniesz podczas pracy nad dowolnym ćwiczeniem przedstawionym w książce, zrób sobie przerwę i zajmij się czymś innym. Jeżeli później nadal nie będziesz umiał poradzić sobie z problemem, przejrzyj odpowiednie fragmenty rozdziału. Jeśli mimo to wciąż będziesz stał w miejscu i uznasz, że potrzebujesz pomocy z zewnątrz, przydatne wskazówki znajdziesz w dodatku C.*

## ZRÓB TO SAM

Utwórz oddzielne programy dla każdego z poniższych ćwiczeń. Poszczególne programy umieszczaj w plikach o nazwach zgodnych ze standardowymi konwencjami Pythona, czyli zawierających jedynie małe litery i znaki podkreślenia, na przykład `prosty_komunikat.py` i `proste_komunikaty.py`.

**2.1. Prosty komunikat.** Umieść komunikat w zmiennej, a następnie wyświetl go.

**2.2. Proste komunikaty.** Umieść komunikat w zmiennej, a następnie go wyświetl. Później zmień wartość zmiennej tak, aby zawierała inny komunikat, i ten nowy komunikat również wyświetl.

# Ciągi tekstowe

Ponieważ większość programów definiuje i zbiera pewnego rodzaju dane, a następnie przeprowadza na nich użyteczne operacje, pomocne jest klasyfikowanie różnych typów tych danych. Poznawanie typów danych zaczynamy od ciągu tekstopowego. Na początku ciągi tekstowe wydają się niezwykle proste, można je jednak wykorzystać na wiele różnych sposobów.

Wspomniany *ciąg tekstopowy* to po prostu seria znaków. Wszystko to, co zostało ujęte w znaki cytowania, jest uznawane w Pythonie za ciąg tekstopowy. Ciąg tekstopowy może być ujęty zarówno w cudzysłów, jak i apostrofy, na przykład:

---

```
"To jest ciąg tekstopowy."  
'To również jest ciąg tekstopowy.'
```

---

Taka elastyczność pozwala na użycie dosłownych znaków cytowania wewnętrz ciągów tekstopowych:

---

```
'Powiedziałem koleżce: "Python to mój ulubiony język programowania!".'  
"Nazwa języka 'Python' pochodzi od Monty Pythona, a nie węża."  
"Dla programisty Johna O'Hary jedną z zalet Pythona jest jego wszechstronność  
→ i oddana mu społeczność."
```

---

Przeanalizujmy różne sposoby używania ciągów tekstopowych.

## Zmiana wielkości liter ciągu tekstopowego za pomocą metod

Jednym z najprostszych zadań, które można wykonać względem ciągu tekstopowego, jest zmiana wielkości liter w pojawiających się w nim słowach. Spójrz na poniższy fragment kodu i spróbuj powiedzieć, jaki będzie wynik jego działania.

## Plik name.py:

---

```
name = "jan kowalski"  
print(name.title())
```

---

Umieść kod w pliku o nazwie *name.py*, a następnie uruchom go. Powinieneś otrzymać następujące dane wyjściowe.

---

Jan Kowalski

---

W tym przykładzie zapisany małymi literami ciąg tekstowy jan kowalski został umieszczony w zmiennej `name`. Wywołanie metody `title()` znajduje się po nazwie zmiennej `name` w metodzie `print()`. Wspomniana tutaj *metoda* to operacja, którą Python może przeprowadzić na pewnych danych. Kropka po nazwie zmiennej `name` w wywołaniu `name.title()` wskazuje Pythonowi, że metoda `title()` ma zostać wykonana względem zmiennej `name`. Po nazwie każdej metody znajduje się para nawiasów okrągłych, ponieważ metody do prawidłowego działania często potrzebują informacji dodatkowych. Te informacje są umieszczane wewnątrz nawiasu. Metoda `title()` nie wymaga żadnych informacji dodatkowych, więc nawias pozostaje pusty.

Działanie metody `title()` polega na zmianie pierwszej litery każdego słowa na wielką. Jest to przydatna operacja, ponieważ często zachodzi potrzeba potraktowania imienia jako fragmentu informacji. Na przykład tworzysz program i chcesz, aby rozpoznawał on dane wejściowe w postaci Jan, JAN i jan jako to samo imię i wyświetlał je jako Jan.

Aby zmienić wielkość liter, możesz wykorzystać jeszcze dwie inne metody. Na przykład istnieje możliwość zmiany wielkości wszystkich liter na małe lub duże, jak pokazalem w poniższym fragmencie kodu:

---

```
name = "Jan Kowalski"  
print(name.upper())  
print(name.lower())
```

---

Powinieneś otrzymać następujące dane wyjściowe:

---

JAN KOWALSKI  
jan kowalski

---

Metoda `lower()` jest szczególnie użyteczna podczas przechowywania danych. Wielokrotnie zdarza się, iż nie można ufać użytkownikom, że podali właściwą wielkość liter. Dlatego też najlepiej skonwertować te ciągi tekstowe na zapisane małymi literami i przechowywać je w takiej postaci. Gdy zajdzie konieczność wyświetlania informacji, można użyć takiej wielkości liter, która będzie miała sens dla poszczególnych ciągów tekstowych.

## Używanie zmiennych w ciągach tekstowych

Bardzo często użyteczne jest stosowanie zmiennych wewnętrz ciągów tekstowych. Na przykład chcesz przechowywać imię i nazwisko w oddzielnych zmiennych, a następnie łączyć je w celu wyświetlenia pełnego imienia i nazwiska:

Plik full\_name.py:

---

```
first_name = "jan"
last_name = "kowalski"
full_name = f"{first_name} {last_name}" ❶
print(full_name)
```

---

Aby wstawić wartość zmiennej do ciągu tekstu, należy tuż przed znakiem cytowania umieścić literę `f` ❶. Nazwa każdej zmiennej, której wartość ma się znaleźć w ciągu tekstu, musi zostać umieszczona w nawiasie klamrowym. Podczas wyświetlania ciągu tekstu poszczególne zmienne będą zastąpione odpowiadającymi im wartościami.

To jest przykład tzw. *ciągu tekstuowego f* (ang. *f-string*). Litera `f` pochodzi od słowa *format*, ponieważ Python formatuje ciąg tekstowy, zastępując podaną w nawiasie klamrowym nazwę zmiennej jej wartością. Oto dane wyjściowe wygenerowane przez omówiony fragment kodu:

---

```
jan kowalski
```

---

Ciągi tekstowe f mają wiele możliwości. Na przykład można je wykorzystać do przygotowywania pełnych komunikatów na podstawie informacji przechowywanych w zmiennych. Spójrz na przedstawiony poniżej przykład:

---

```
first_name = "jan"
last_name = "kowalski"
full_name = f"{first_name} {last_name}"
print(f"Witaj, {full_name.title()}")❶
```

---

W powyższym fragmencie kodu pełne imię i nazwisko jest używane w wierszu ❶, w zdaniu zawierającym powitanie użytkownika. Metoda `title()` została wykorzystana w celu odpowiedniego sformatowania imienia i nazwiska. Dane wyjściowe z omówionego fragmentu kodu to prosty, choć jednocześnie elegancki komunikat powitania:

---

```
Witaj, Jan Kowalski!
```

---

Oczywiście istnieje możliwość użycia ciągów tekstowych f do przygotowania komunikatu, a następnie umieszczenia go w zmiennej, na przykład tak jak poniżej:

---

```
first_name = "jan"
last_name = "kowalski"
full_name = f"{first_name} {last_name}"
message = f"Witaj, {full_name.title()}!"❶
print(message) ❷
```

---

Powyższy fragment kodu również wyświetla komunikat `Witaj, Jan Kowalski!`, ale całe powitanie jest przechowywane w zmiennej (patrz wiersz ❶), co sprawia, że wywołanie polecenia `print()` ma znacznie prostszą postać (patrz wiersz ❷).

## Dodawanie białych znaków do ciągów tekstowych za pomocą tabulatora i znaku nowego wiersza

W programowaniu *biały znak* oznacza każdy niedrukowany znak, taki jak spacja, tabulator lub znak końca wiersza. Za pomocą białych znaków można sformatować dane wyjściowe tak, aby stały się dla użytkowników łatwiejsze do odczytania.

W celu dodania tabulatora do tekstu należy użyć sekwencji znaków `\t`:

---

```
>>> print("Python")
Python
>>> print("\tPython")
    Python
```

---

Jeżeli chcesz dodać znak nowego wiersza do ciągu tekstowego, użyj sekwencji `\n`:

---

```
>>> print("Języki:\nPython\nC\nJavaScript")
Języki:
Python
C
JavaScript
```

---

Oczywiście istnieje możliwość połączenia w jednym ciągu tekstowym tabulatorów i znaków nowego wiersza. Sekwencja `\n\t` nakazuje Pythonowi przejście do nowego wiersza i rozpoczęcie go od znaku tabulatora. W poniższym fragmencie kodu pokazałem, jak można użyć jednowierszowego ciągu tekstopo- wego do wygenerowania czterech wierszy danych wyjściowych:

---

```
>>> print("Języki:\n\tPython\n\tC\n\tJavaScript")
Języki:
    Python
    C
    JavaScript
```

---

Znaki nowego wiersza i tabulatory będą niezwykle użyteczne w dwóch następnych rozdziałach, w których zacznijemy generować wiele wierszy danych wyjściowych za pomocą jedynie kilku wierszy kodu źródłowego.

## Usunięcie białych znaków

Dodatkowy biały znak może wprowadzać zamieszanie w programach. Dla programisty ciągi tekstowe 'python' i 'python ' wyglądają praktycznie identycznie, natomiast z perspektywy programu to są dwa różne ciągi tekstowe. Python wykrywa dodatkową spację w 'python ' i uznaje ją za ważną, o ile nie wskażeś, że jest inaczej.

Bardzo ważne jest uwzględnianie białych znaków, ponieważ często będzie trzeba porównywać dwa ciągi tekstowe, aby ustalić, czy są jednakowe. Tego rodzaju operacja jest na przykład przeprowadzana, kiedy sprawdza się nazwy użytkowników logujących się w witrynie internetowej. Dodatkowy biały znak może wprowadzać zamieszanie także w prostszych sytuacjach. Na szczęście Python niezwykle ułatwia usuwanie dodatkowych białych znaków z danych wprowadzanych przez użytkowników.

Python może sprawdzać obecność białych znaków po lewej i prawej stronie ciągu tekstu. Aby mieć pewność, że po prawej stronie ciągu tekstu nie ma białych znaków, należy użyć metody `rstrip()`:

```
>>> favorite_language = 'python ' ❶
>>> favorite_language ❷
'python '
>>> favorite_language.rstrip() ❸
'python'
>>> favorite_language ❹
'python '
```

Wartość przechowywana w zmiennej `favorite_language` zawiera dodatkowy biały znak na końcu ciągu tekstu (patrz wiersz ❶). Kiedy w sesji powłoki poprosisz Pythona o podanie wartości wymienionej zmiennej, możesz zobaczyć biały znak na końcu (patrz wiersz ❷). Gdy metoda `rstrip()` działa na zmiennej `favorite_language` tak jak w wierszu ❸, ten dodatkowy biały znak zostaje usunięty. Jednak to usunięcie jest tylko tymczasowe. Gdy ponownie sprawdzisz wartość wymienionej zmiennej, zobaczyś, że ciąg tekstowy ma dokładnie tę postać, w jakiej został wprowadzony, czyli łącznie z białym znakiem (patrz wiersz ❹).

Aby trwale usunąć biały znak z ciągu tekstu, konieczne jest umieszczenie pozbawionej go wartości ponownie w tej samej zmiennej:

```
>>> favorite_language = 'python '
>>> favorite_language = favorite_language.rstrip() ❶
>>> favorite_language
'python'
```

Aby trwale usunąć biały znak z ciągu tekstopowego, pozbywamy się go z prawej strony, a następnie wartość ponownie umieszczamy w pierwotnej zmiennej, tak jak pokazałem w wierszu ①. Zmiana wartości zmiennej i umieszczenie nowej wartości z powrotem w pierwotnej zmiennej jest często stosowanym rozwiąza niem w programowaniu. W ten sposób wartość zmiennej może być modyfikowana podczas wykonywania programu, lub też w odpowiedzi na dane wejściowe użytkownika.

Do usunięcia białego znaku z lewej strony ciągu tekstopowego służy metoda `lstrip()`, natomiast pozbycie się białych znaków z obu stron ciągu tekstopowego umożliwia metoda `strip()`:

---

```
>>> favorite_language = ' python ' ❶
>>> favorite_language.rstrip() ❷
' python'
>>> favorite_language.lstrip() ❸
'python '
>>> favorite_language.strip() ❹
'python'
```

---

W omawianym przykładzie rozpoczynamy od wartości zawierającej białe znaki na początku i końcu ciągu tekstopowego (patrz wiersz ❶). Następnie usuwamy nadmiarowe znaki z prawej strony (patrz wiersz ❷), lewej (patrz wiersz ❸) oraz z obu (patrz wiersz ❹). Eksperymenty z funkcjami usuwającymi białe znaki pomogą Ci w nabyciu większej wprawy w przeprowadzaniu operacji na ciągach tekstopowych. W rzeczywistych programach przedstawione powyżej metody przeznaczone do usuwania białych znaków są najczęściej stosowane do oczyszczania danych wejściowych, zanim zaczną być one przechowywane w programie.

## Usunięcie prefiku

Inne często wykonywane zadanie podczas pracy z ciągami tekstopowymi to usunięcie prefiku. Na przykład prefiksem adresu URL najczęściej jest `https://`. Założmy, że chcesz usunąć ten prefiks, aby skoncentrować się jedynie na tej części adresu URL, którą użytkownicy muszą wpisać w pasku adresu przeglądarki WWW. Zobacz, jak można to zrobić w kodzie Pythona:

---

```
>>> no_starch_url = 'https://nostarch.com'
>>> no_starch_url.removeprefix('https://')
'nostarch.com'
```

---

Wystarczy podać nazwę zmiennej, kropkę i nazwę metody `removeprefix()`. W nawiązaniu trzeba podać nazwę prefiku, który ma zostać usunięty z ciągu tekstopowego.

Podobnie jak metoda, która służy do usuwania białych znaków, także metoda `removeprefix()` pozostawia nietknięty pierwotny ciąg tekstowy. Jeżeli chcesz zachować nową wartość bez prefiksu, możesz ją przypisać tej samej bądź nowej zmiennej:

```
>>> simple_url = no_starch_url.removeprefix('https://')
```

Gdy w pasku adresu przeglądarki WWW zobaczysz adres URL bez prefiku `https://`, wskazuje to na użycie w przeglądarce metody typu `removeprefix()`.

## Unikanie błędów składni w ciągach tekstowych

Jednym z najczęstszych rodzajów błędów, z jakim można się spotkać, jest błąd składni. Wspomniany *błąd składni* występuje, gdy Python nie potrafi rozpoznać fragmentu kodu jako prawidłowego kodu Pythona. Na przykład użycie apostrofu w ciągu tekstowym ujętym w apostrofach spowoduje wygenerowanie błędu. Tak się dzieje, ponieważ Python wszystko, co jest między pierwszym apostrofem i dodatkowym znakiem apostrofu użyтыm w tekście, interpretuje jako ciąg tekstowy. Następnie pozostała część tekstu próbuje zinterpretować jako kod Pythona, co prowadzi do błędów.

Poniżej pokazałem, jak należy prawidłowo używać apostrofów i cudzysłowu. Zapisz program w pliku o nazwie *apostrophe.py*, a następnie uruchom go.

*Plik apostrophe.py:*

```
message = "Dla programisty Johna O'Hary jedną z zalet Pythona jest jego  
wszechstronność i oddana mu społeczność."  
print(message)
```

Ponieważ apostrof znajduje się wewnętrz ciągu tekstowego wziętego w cudzysłów, interpreter Pythona nie ma żadnych problemów z prawidłowym jego odczytaniem:

```
Dla programisty Johna O'Hary jedną z zalet Pythona jest jego wszechstronność  
i oddana mu społeczność.
```

Jednak jeżeli użyjesz jedynie apostrofów, Python nie będzie umiał prawidłowo określić miejsca zakończenia ciągu tekstowego:

```
message = 'Dla programisty Johna O'Hary jedną z zalet Pythona jest jego  
wszechstronność i oddana mu społeczność.'  
print(message)
```

Powinieneś otrzymać następujące dane wyjściowe:

```
File "apostrophe.py", line 1
    message = 'Dla programisty Johna O'Hary jedną z zalet Python'a jest jego
              ←wszechstronność i oddana mu społeczność.'
                           ^
  ①
SyntaxError: unterminated string literal (detected at line 1)
```

Informacje przedstawione w danych wyjściowych wskazują na istnienie błędu (patrz wiersz ①) tuż po drugim apostrofie. Tego rodzaju błąd składni oznacza, że interpreter nie rozpoznał pewnych fragmentów jako prawidłowego kodu Pythona. Dlatego interpreter uważa, że problem może mieć związek z niepoprawnym użyciem znaków cytowania w ciągu tekstu. Błędy mogą mieć wiele źródeł — te najczęściej występujące będą pokazywały na bieżąco, w miarę ich pojawiania się. Podczas nauki tworzenia poprawnego kodu Pythona najczęściej będziesz się spotykać z błędami składni. Są one zaliczane do najbardziej uciążliwych, ponieważ próby ich wychwytyowania i usunięcia mogą sprawiać wiele trudności i mocno frustrować. Gdy natrafisz na błąd, którego nie będziesz umiał naprawić, wówczas zajrzyj do dodatku C — znajdziesz tam pewne wskazówki i podpowiedzi, co dalej zrobić.

**UWAGA** *Oferowana przez edytor tekstu funkcja kolorowania składni powinna pomóc w szybkim wychwytywaniu niektórych błędów składni jeszcze podczas tworzenia programów. Jeżeli zauważysz oznaczenie kodu Pythona jako słowa napisanego w języku polskim, lub też na odwrót, prawdopodobnie zapomniałeś gdzieś w pliku umieścić znak cytowania.*

## ZRÓB TO SAM

Przedstawione poniżej ćwiczenia zapisz w oddzielnych plikach o nazwach takich jak *wielkosc\_liter\_w\_imionach.py*. Jeżeli gdzieś utkniesz, zrób sobie przerwę lub zapoznaj się z podpowiedziami zamieszczonymi w dodatku C.

**2.3. Osobiste powitanie.** Zapisz w zmiennej imię osoby, a następnie wyświetl dla niej komunikat powitania. Komunikat powinien być prosty, na przykład „Witaj, Eryk! Czy chcesz dzisiaj poznawać Pythona?”.

**2.4. Wielkość liter w imionach.** Zapisz w zmiennej imię osoby, a następnie wyświetl je za pomocą małych liter, wielkich liter oraz z użyciem wielkiej litery jako pierwszej litery imienia.

**2.5. Stawny cytat.** Odszukaj cytat sławnej osoby, którą cenisz. Wyświetl ten cytat wraz z imieniem i nazwiskiem jego autora. Wygenerowane dane wyjściowe powinny wyglądać tak, jak pokazałem poniżej, łącznie ze znakami cytowania: *Albert Einstein powiedział kiedyś: "Osoba, która nigdy nie popełniła błędu, jest kimś, kto nigdy nie próbował niczego nowego".*

**2.6. Sławny cytat 2.** Powtórz ćwiczenie 2.5, ale tym razem imię i nazwisko autora cytatu umieść w zmiennej o nazwie `famous_person`. Następnie przygotuj komunikat i umieść go w nowej zmiennej o nazwie `message`. Na koniec wyświetl komunikat.

**2.7. Usunięcie białych znaków z imienia.** Zapisz w zmiennej imię osoby wraz z pewnymi białymi znakami na początku i końcu imienia. Upewnij się, że co najmniej raz użyłeś sekwencji `\t` i `\n`.

Wyświetl to imię wraz ze znakami odstępu. Następnie wyświetl je jeszcze trzy razy, ale za każdym razem wykorzystaj jedną z metod przeznaczonych do usuwania białych znaków: `lstrip()`, `rstrip()` i `strip()`.

**2.8. Rozszerzenie pliku.** Python ma metodę o nazwie `removeprefix()`, która działa podobnie jak `removeprefix()`. Zmiennej `filename` przypisz wartość `'python_notes.txt'`. Następnie za pomocą metody `removeprefix()` wyświetl nazwę pliku bez rozszerzenia, podobnie jak to ma miejsce w przeglądarkach plików.

# Liczby

Liczby są dość często używane w programowaniu, na przykład do przechowywania wyniku w grze, przedstawienia danych w wizualizacjach czy przechowywania informacji w aplikacjach internetowych. Sposób, w jaki Python traktuje te liczby, zależy od tego, jak są one używane. Najpierw zobaczysz, jak Python zarządza liczbami całkowitymi, ponieważ praca z nimi jest najłatwiejsza.

## Liczby całkowite

W Pythonie można dodawać (+), odejmować (-), mnożyć (\*) i dzielić (/) liczby całkowite:

```
>>> 2 + 3  
5  
>>> 3 - 2  
1  
>>> 2 * 3  
6  
>>> 3 / 2  
1.5
```

W przypadku sesji powłoki Python po prostu wyświetla wynik operacji. Do przedstawienia wykładnika potęgi w Pythonie są używane dwa znaki mnożenia:

```
>>> 3 ** 2  
9  
>>> 3 ** 3  
27
```

```
>>> 10 ** 6  
1000000
```

---

Oczywiście Python obsługuje kolejność wykonywania działań, więc jedno wyrażenie może zawierać ich wiele. Do zmiany kolejności działań można użyć nawiasów, co pozwoli Pythonowi na wykonywanie ich według ustalonego porządku. Spójrz na przedstawiony poniżej przykład:

```
>>> 2 + 3*4  
14  
>>> (2 + 3) * 4  
20
```

---

Spacje w powyższych przykładach nie mają wpływu na sposób obliczenia wyrażenia przez Pythona. Mają po prostu pomóc człowiekowi, kiedy ten będzie analizował kod, ustalić, która operacja ma wyższy priorytet.

## Liczby zmiennoprzecinkowe

Każda liczba z przecinkiem dziesiętnym jest w Pythonie określana mianem *liczby zmiennoprzecinkowej* (ang. *float*). Ten termin jest stosowany w większości języków programowania i odnosi się do tego, że przecinek dziesiętny może pojawiać się na dowolnej pozycji w liczbie. Każdy język programowania musi być z ogromną ostrożnością zaprojektowany do prawidłowej obsługi liczb zmiennoprzecinkowych, aby ich zachowanie było prawidłowe niezależnie od miejsca występowania przecinka dziesiętnego.

W większości sytuacji nie trzeba przejmować się zachowaniem liczb zmiennoprzecinkowych. Wystarczy po prostu wprowadzić liczby przeznaczone do użycia, a Python prawdopodobnie wygeneruje oczekiwany wynik:

```
>>> 0.1 + 0.1  
0.2  
>>> 0.2 + 0.2  
0.4  
>>> 2 * 0.1  
0.2  
>>> 2 * 0.2  
0.4
```

---

Trzeba mieć jednak świadomość, że czasami w wygenerowanych danych wyjściowych można otrzymać inną niż oczekiwania ilość cyfr po przecinku dziesiętnym:

```
>>> 0.2 + 0.1  
0.30000000000000004  
>>> 3 * 0.1  
0.30000000000000004
```

---

Taka sytuacja może pojawić się we wszystkich językach programowania. Python stara się przedstawić liczby w maksymalnie precyzyjnej postaci, co jednak czasami jest trudne ze względu na sposób, w jaki komputer wewnętrznie przedstawia liczby. Na razie po prostu zignoruj dodatkowe cyfry po przecinku dziesiętnym. W części drugiej książki, gdy przejdziemy do projektów, dowiesz się, jak można radzić sobie z dodatkowymi cyframi po przecinku dziesiętnym.

## Liczby całkowite i zmiennoprzecinkowe

Gdy dzielisz dwie dowolne liczby, nawet jeśli obie są liczbami całkowitymi dającymi w wyniku również liczbę całkowitą, wynikiem zawsze będzie liczba zmiennoprzecinkowa:

---

```
>>> 4/2  
2.0
```

---

W przypadku połączenia liczb całkowitych i zmiennoprzecinkowych w innej dowolnej operacji wynikiem także będzie liczba zmiennoprzecinkowa:

---

```
>>> 1 + 2.0  
3.0  
>>> 2 * 3.0  
6.0  
>>> 3.0 ** 2  
9.0
```

---

Python domyślnie stosuje typ liczby zmiennoprzecinkowej w każdej operacji używającej takich liczb, nawet jeśli wynikiem operacji jest liczba całkowita.

## Znaki podkreślenia w liczbach

Przy zapisywaniu ogromnych liczb cyfry można grupować za pomocą znaków podkreślenia, aby w ten sposób poprawić czytelność liczby:

---

```
>>> universe_age = 14_000_000_000
```

---

W trakcie wyświetlania liczby zdefiniowanej z użyciem znaków podkreślenia Python pokaże jedynie cyfry:

---

```
>>> print(universe_age)  
14000000000
```

---

Python ignoruje znaki podkreślenia podczas przechowywania takich wartości. Nawet jeśli nie będziesz grupować cyfr trójkami, wartość i tak pozostanie nienaruszona. Z perspektywy Pythona wartość 1000 jest dokładnie taka sama jak 1\_000 lub

10\_00. Ta funkcjonalność sprawdza się w przypadku liczb całkowitych i zmien- noprzecinkowych.

## Wiele przypisań

W pojedynczym wierszu kodu można przypisać wartość więcej niż jednej zmiennej. Taka możliwość pozwala skrócić programy i ułatwia ich odczyt. Tę technikę będziesz najczęściej stosować podczas inicjalizacji zbioru elementów.

Zobacz, jak można zainicjalizować wartością 0 zmienne x, y i z:

---

```
>>> x, y, z = 0, 0, 0
```

---

Poszczególne nazwy zmiennych muszą być rozdzielone przecinkami, po- dośnie jak wartości — Python przypisze każdą wartość odpowiedniej zmiennej. O ile liczba wartości odpowiada liczbie zmiennych, Python prawidłowo je do siebie dopasuje.

## Stałe

*Stała* przypomina zmienną, której wartość nie ulega zmianie w trakcie całego cyklu życia programu. Python nie ma wbudowanego typu przeznaczonego dla stałej. Konwencją stosowaną przez programistów jest używanie tylko wielkich liter do wskazania zmiennej, która ma być traktowana jak stała i nigdy nie zmieniać wartości:

---

```
MAX_CONNECTIONS = 5000
```

---

Gdy w kodzie chcesz traktować zmienną jako stałą, upewnij się, że jej nazwa została zapisana za pomocą tylko wielkich liter.

### ZRÓB TO SAM

**2.9. Liczba osiem.** Zapisz operacje dodawania, odejmowania, mnożenia i dzielenia, których wynikiem będzie liczba 8. Upewnij się, że użyłeś funkcji print(), aby wyświetlić wyniki. Powinieneś utworzyć cztery wiersze, które będą wyglądały tak jak ten poniżej:

---

```
print(5 + 3)
```

---

Wygenerowane dane wyjściowe powinny składać się po prostu z czterech wierszy wraz z liczbą 8 pojawiającą się jednokrotnie w każdym z nich.

**2.10. Ulubiona liczba.** Umieść w zmiennej ulubioną liczbę. Następnie używając tej zmiennej, utwórz komunikat ujawniający tę ulubioną liczbę. Wyświetl ten komunikat.

# Komentarze

Komentarze to niezwykle użyteczna funkcja w większości języków programowania. Wszystko to, co dotąd umieszczaliśmy w programach, jest kodem Pythona. Gdy tworzone programy są dłuższe i bardziej skomplikowane, wówczas warto umieszczać w nich notatki opisujące ogólne podejście do rozwiązywanego problemu. *Komentarz* pozwala na umieszczanie w programach notatek zapisanych na przykład w języku polskim.

## Jak można utworzyć komentarz?

Komentarz w języku Python jest oznaczany za pomocą znaku hash. Wszystko to, co znajduje się po znaku #, jest ignorowane przez interpreter Pythona. Spójrz na przedstawiony poniżej przykład.

Plik comment.py:

```
# Przywitaj się ze wszystkimi.  
print("Witajcie, programiści Pythona!")
```

Python zignoruje pierwszy wiersz i wykona kod zdefiniowany w drugim:

```
Witajcie, programiści Pythona!
```

## Jakiego rodzaju komentarze należy tworzyć?

Podstawowym powodem umieszczania komentarzy w kodzie źródłowym jest wyjaśnienie przeznaczenia i sposobu działania danego fragmentu kodu. Kiedy jesteś w trakcie pracy nad projektem, doskonale rozumiesz wszystkie jego aspekty. Jednak jeśli powrócisz do projektu po pewnym czasie, prawdopodobnie zapomnisz już o pewnych szczegółach. Wprawdzie zawsze można poświęcić nieco czasu na przeanalizowanie kodu i ustalenie sposobu działania jego poszczególnych fragmentów, ale wcześniejsze przygotowanie dobrych komentarzy pomaga zaoszczędzić ten czas, ponieważ można po prostu przeczytać zapisane jasno i wyraźnie podsumowanie zastosowanego w kodzie ogólnego podejścia do danego problemu.

Jeżeli chcesz się stać profesjonalnym programistą lub współpracować z innymi, musisz umieszczać w kodzie jasne i czytelne komentarze. Obecnie większość oprogramowania jest tworzona przez wiele osób, na przykład grupę pracowników w jednej firmie lub grupę osób, które po prostu wspólnie pracują nad projektem. Doskonale wyszkolony programista oczekuje istnienia komentarzy w kodzie, więc warto zacząć je dodawać już teraz. Tworzenie w kodzie jasnych i spójnych komentarzy to jeden z najlepszych nawyków, jakie można w sobie wyrobić.

Kiedy próbujesz ustalić, czy należy utworzyć komentarz, zadaj sobie pytanie, czy konieczne było rozważenie kilku możliwych rozwiązań danego problemu, zanim znalazłeś to właściwe. Jeżeli odpowiedź jest twierdząca, wstaw komentarz dotyczący tego rozwiązania. Znacznie łatwiej jest później usunąć dodatkowy komentarz, niż

powrócić do programu i tworzyć komentarze dla słabo udokumentowanego kodu źródłowego. Od teraz będę używał komentarzy w przykładowych fragmentach kodu przedstawianych w książce, aby pomóc w wyjaśnieniu znaczenia poszczególnych sekcji.

### ZRÓB TO SAM

**2.11. Dodawanie komentarzy.** Wybierz dwa utworzone dotąd programy i umieść w nich przynajmniej po jednym komentarzu. Jeżeli nie masz nic ciekawego do napisania z powodu prostoty dotychczasowych programów, wpisz na początek pliku po prostu imię i datę. Następnie dodaj jedno zdanie opisujące przeznaczenie danego programu.

## Zen Pythona

Doświadczeni programiści Pythona będą zachęcali Cię, abyś unikał komplikowania kodu i stawał na prostotę, gdy to tylko możliwe. Filozofia społeczności Pythona jest zawarta w Zen Pythona opracowanym przez Tim'a Petersa. Dostęp do krótkiego zbioru reguł dotyczących tworzenia dobrego kodu w Pythonie możesz uzyskać, używając w interpreterze polecenia `import this`. Nie będę tutaj przedstawał całego Zen Pythona, ale zaprezentuję kilka wierszy pomagających zrozumieć, dlaczego te reguły są tak ważne dla początkujących programistów Pythona.

---

```
>>> import this
```

The Zen of Python, by Tim Peters

---

```
Beautiful is better than ugly. # Piękno jest lepsze od brzydoty.
```

---

Programiści Pythona są przekonani, że kod może być piękny i elegancki. Podczas programowania zadaniem programistów jest rozwiązywanie problemów. Programiści zawsze byli darzeniem szacunkiem za dobre przygotowane, efektywne i nawet piękne rozwiązania problemów. Gdy lepiej poznasz Pythona i zaczniesz go używać do tworzenia większej ilości kodu, pewnego dnia ktoś może spojrzeć Ci przez ramię i zachwycić się „Wow, to jest naprawdę piękny kod!”.

---

```
Simple is better than complex. # Prostota jest lepsza od zawiłości.
```

---

Jeżeli masz wybór między prostym i skomplikowanym rozwiązaniem, a oba się sprawdzają, wybieraj prostsze. Twój kod będzie łatwiejszy w późniejszej obsłudze, a ponadto zarówno Tobie, jak i innym łatwiej będzie go wykorzystać jako podstawę do dalszej rozbudowy.

---

`Complex is better than complicated.` # Złożoność jest lepsza od skomplikowania.

---

Rzeczywistość jest skomplikowana i czasami proste rozwiązanie problemu okazuje się nieosiągalne. W takim przypadku zdecyduj się na najprostsze rozwiązanie, które działa.

---

`Readability counts.` # Przejrzystość ma znaczenie.

---

Nawet jeśli kod jest skomplikowany, postaraj się, aby pozostał czytelny. Kiedy pracujesz nad projektem zawierającym skomplikowany kod, skoncentruj się na utworzeniu przydatnych komentarzy objaśniających jego działanie.

---

`There should be one-- and preferably only one --obvious way to do it.`  
# Powinien istnieć jeden – i najlepiej tylko jeden – oczywisty sposób wykonania danego zadania.

---

Jeżeli dwóch programistów Pythona zostanie poproszonych o rozwiązanie tego samego problemu, powinni przygotować bardzo podobny kod. To oczywiście nie oznacza braku miejsca na kreatywność w programowaniu, a wręcz przeciwnie! Jednak w większości sytuacji programowanie oznacza używanie typowych rozwiązań dla prostych problemów istniejących w ramach większego, bardziej kreatywnego projektu. Podstawy tworzonych przez Ciebie programów powinny mieć sens dla innych programistów Pythona.

---

`Now is better than never.` # Teraz jest lepsze od nigdy.

---

Móglbyś poświęcić resztę życia na poznawanie wszystkich zawiłości Pythona oraz ogólnie programowania, ale wówczas nie ukończyłbyś żadnego projektu. Nie staraj się stworzyć perfekcyjnego kodu, lecz skoncentruj się na pisaniu kodu, który działa. Dopiero później podejmij decyzję, czy należy usprawnić kod danego projektu, czy raczej przejść do nowego.

Gdy będziesz kontynuował naukę w następnym rozdziale i przejdziesz do bardziej zaawansowanych tematów, pamiętaj o filozofii prostoty i przejrzystości. Doświadczeni programiści będą wówczas bardziej doceniać Twój kod oraz częściej przekazywać Ci swoje uwagi do niego i częściej współpracować z Tobą podczas realizacji interesujących projektów.

## ZRÓB TO SAM

**2.12. Zen Pythona.** Wydaj polecenie `import this` w powłoce Pythona i przejrzyj reguły, które nie zostały przedstawione w tym rozdziale.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak pracować ze zmiennymi. Przekonałeś się, jak używać jasnych i przejrzystych nazw zmiennych, a także jak usuwać błędy związane z nazwami oraz składnią, gdy takowe się pojawią. Poznałeś ciągi tekstowe, zobaczyłeś, jak wyświetlać je z użyciem małych i wielkich liter, a także jak zmienić pierwszą literę każdego słowa na wielką. Nauczyłeś się stosować białe znaki do eleganckiego wyświetlania danych wyjściowych oraz pozbywać się tych niechcianych białych znaków z różnych miejsc ciągu tekstopowego. Zacząłeś pracę z liczbami całkowitymi i zmiennoprzecinkowymi, dowiedziałeś się o pewnym nieoczekiwany zachowaniu Pythona, na które należy zwrócić uwagę podczas używania wartości liczbowych. Ponadto zobaczyłeś, jak tworzyć jasne i przejrzyste komentarze, aby kod stał się łatwiejszy do odczytania zarówno dla Ciebie, jak i innych programistów. Na końcu poznaleś trochę filozofię zachowania maksymalnej prostoty kody, gdy tylko to możliwe.

W rozdziale 3. zajmiemy się tematem przechowywania kolekcji informacji w zmiennych określanych mianem *list*. Zobaczysz, jak przeprowadzać iteracje przez listę oraz operować na znajdującymi się w niej danych.

# 3

## Wprowadzenie do list



W TYM I W NASTĘPNYM ROZDZIALE POZNASZ LISTY ORAZ ROZPOCZNESZ PRACĘ Z ELEMENTAMI UMIESZCZANYMI NA LIŚCIE. LISTA POZWALA PRZECIĘTNIOWAĆ ZBIÓR INFORMACJI W JEDNYM MIEJSCU NIEZALEŻNIE OD TEGO, czy zawiera jedynie kilka elementów czy miliony. Lista to jedna z najpotężniejszych funkcji Pythona, łatwo dostępna dla nowych programistów tego języka i wykorzystująca wiele ważnych koncepcji programowania.

### Czym jest lista?

*Lista* to kolekcja elementów ułożonych w określonej kolejności. Możesz utworzyć listę zawierającą litery alfabetu, cyfry od 0 do 9, imiona wszystkich członków rodziny itd. Na liście można umieścić cokolwiek, a jej poszczególne elementy nie muszą być w żaden sposób ze sobą powiązane. Ponieważ lista z reguły zawiera więcej niż tylko jeden element, dobrze jest, aby jej nazwa była sformułowana w liczbie mnogiej, na przykład `litery`, `cyfry`, `imiona`.

W Pythonie nawias kwadratowy (`[]`) wskazuje listę, a jej poszczególne elementy są rozdzielone przecinkami. Poniżej przedstawiłem przykład prostej listy zawierającej kilka rodzajów rowerów.

*Plik bicycles.py:*

---

```
bicycles = ['trekingowy', 'górski', 'miejski', 'szosowy']
print(bicycles)
```

---

Jeżeli poprosisz Pythona o wyświetlenie listy, interpreter zwróci jej reprezentację w całości, wraz z nawiasami kwadratowymi:

---

```
['trekingowy', 'górski', 'miejski', 'szosowy']
```

---

Ponieważ to nie jest rodzaj danych wyjściowych oczekiwanych przez użytkowników, zobaczysz zaraz, jak można uzyskać dostęp do poszczególnych elementów listy.

## Uzyskanie dostępu do elementów listy

Lista jest uporządkowaną kolekcją i dlatego dostęp do dowolnego jej elementu można uzyskać przez podanie jego położenia, czyli tak zwanego *indeksu*. Aby uzyskać dostęp do elementu listy, należy podać jej nazwę, a następnie indeks ujęty w nawias kwadratowy.

Na przykład w poniższym fragmencie kodu pobieramy pierwszy rodzaj roweru z listy `bicycles`:

---

```
bicycles = ['trekingowy', 'górski', 'miejski', 'szosowy']
print(bicycles[0])
```

---

Gdy pobieramy tylko jeden element listy, Python zwraca go bez apostrofów i nawiasu kwadratowego:

---

```
trekingowy
```

---

I to jest wynik oczekiwany przez użytkowników — przejrzyste i elegancko sformatowane dane wyjściowe.

Na każdym elemencie listy można wykorzystywać przedstawione w rozdziale 2. metody ciągu tekstowego. Na przykład istnieje możliwość sformatowania elementu '`trekingowy`' przy użyciu metody `title()`:

---

```
bicycles = ['trekingowy', 'górski', 'miejski', 'szosowy']
print(bicycles[0].title())
```

---

W tym przykładzie zostaną wygenerowane takie same dane wyjściowe jak wcześniej, ale słowo '`Trekingowy`' będzie zapisane wielką literą.

## Numeracja indeksu zaczyna się od 0, a nie od 1

Python uznaje, że pierwszy element listy znajduje się w położeniu 0, a nie 1. Takie podejście jest stosowane w większości języków programowania, a powodem jego zastosowania w Pythonie jest sposób, w jaki została przeprowadzona na niskim

poziomie implementacja operacji na listach. Jeżeli otrzymasz nieoczekiwane wyniki, spróbuj ustalić, czy przyczyna nie tkwi w przesunięciu o jeden wartości indeksów.

Drugi element listy ma indeks 1. Stosując ten prosty system, możesz pobrać każdy element listy po prostu przez odjęcie wartości 1 od liczby wskazującej jego położenie na liście. Na przykład aby uzyskać dostęp do czwartego elementu listy, musisz zażądać elementu o indeksie 3.

W poniższym fragmencie kodu pobieramy elementy listy o indeksach 1 i 3:

---

```
bicycles = ['trekingowy', 'górska', 'miejski', 'szosowy']
print(bicycles[1])
print(bicycles[3])
```

---

Kod zwraca więc drugi i czwarty element listy:

---

```
górska
szosowy
```

---

Python oferuje specjalną składnię pozwalającą na łatwe uzyskanie dostępu do ostatniego elementu listy. Jeżeli podasz indeks -1, Python zawsze zwróci ostatni element na liście:

---

```
bicycles = ['trekingowy', 'górska', 'miejski', 'szosowy']
print(bicycles[-1])
```

---

Wartością zwracaną przez powyższy fragment kodu jest 'szosowy'. Przedstawiona składnia jest bardzo użyteczna, ponieważ często zachodzi potrzeba użycia dostępu do ostatniego elementu z listy bez dokładnej wiedzy o liczbie znajdujących się na niej elementów. Ta konwencja rozszerza się także na inne ujemne wartości indeksu. W ten sposób indeks -2 powoduje zwrot drugiego elementu od końca listy, indeks -3 zwraca trzeci element od końca itd.

## Użycie poszczególnych wartości listy

Poszczególne wartości listy mogą być używane w dokładnie taki sam sposób jak każda inna zmienna. Na przykład można zastosować tzw. ciągi tekstowe f w celu utworzenia komunikatu na podstawie wartości listy.

W poniższym fragmencie kodu pobieramy pierwszy rower z listy, a następnie tworzymy komunikat zawierający tę wartość:

---

```
bicycles = ['trekingowy', 'górska', 'miejski', 'szosowy']
message = f"Moim pierwszym rowerem był rower {bicycles[0].title()}."
print(message)
```

---

Tworzymy komunikat z użyciem wartości `bicycles[0]` i umieszczamy go w zmiennej `message`. Dane wyjściowe mają postać prostego zdania wykorzystującego nazwę pierwszego roweru umieszczonego na liście:

---

Moim pierwszym rowerem był rower Trekkingowy.

---

## ZRÓB TO SAM

Wypróbuj poniższe krótkie programy, aby zdobyć nieco doświadczenia w pracy z listami Pythona. Możesz utworzyć nowy katalog dla ćwiczeń w poszczególnych rozdziałach, aby tym samym trochę je uporządkować.

**3.1. Imiona.** Utwórz listę o nazwie `names` i umieść na niej imiona kilku przyjaciół. Wyświetl wszystkie imiona przez uzyskanie dostępu do poszczególnych elementów listy (za każdym razem do jednego).

**3.2. Powitania.** Rozpocznij od listy utworzonej w ćwiczeniu 3.1, ale zamiast samego imiona osoby wyświetl komunikat, który będzie jej dotyczył. Tekst wszystkich komunikatów powinien pozostać taki sam, ale mają być one personalizowane dzięki wykorzystaniu imienia konkretnej osoby.

**3.3. Twoja własna lista.** Zastanów się nad ulubionymi środkami transportu, na przykład motocykl lub samochód, a następnie utwórz listę przechowującą wiele przykładów. Wykorzystaj tę listę do wyświetlenia kilku zdań o jej elementach, na przykład „Chciałbym mieć motocykl Honda”.

# Zmianianie, dodawanie i usuwanie elementów

Większość tworzonych list będzie *dynamiczna*. Oznacza to, że tworzysz listę, a następnie w trakcie działania programu dodajesz do niej elementy i je z niej usuwasz. Na przykład możesz opracować grę, w której zadaniem gracza jest zestrzeliwanie obcych. Początkowy zbiór obcych przechowujesz na liście, a później usuwasz z niej każdego obcego, który został zniszczony przez gracza. Natomiast kiedy na ekranie pojawi się nowy obcy, dodajesz go do listy. Lista obcych będzie zmieniała wielkość w trakcie prowadzonej rozgrywki.

## Modyfikowanie elementów na liście

Składnia przeznaczona do modyfikowania elementów jest podobna do składni uzyskiwania dostępu do elementu listy. Aby zmodyfikować element, należy użyć nazwy listy wraz z indeksem elementu przeznaczonego do zmiany oraz nową wartością, która ma zostać przypisana elementowi.

Na przykład przyjmujemy założenie, że mamy listę motocykli, a pierwszy element to 'honda'. W jaki sposób można zmienić wartość tego elementu?

Plik motorcycles.py:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles[0] = 'ducati'
print(motorcycles)
```

---

Zaczynamy od zdefiniowania listy początkowej, na której pierwszy element to 'honda'. Następnie zmieniamy wartość первого элемента на 'ducati'. Wygenerowane dane wyjściowe faktycznie potwierdzają, że zmienił się tylko ten jeden element na liście:

---

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

---

Oczywiście można zmienić wartość dowolnego elementu listy, a nie tylko pierwszego.

## Dodawanie elementów do listy

Z wielu powodów może wystąpić konieczność dodania nowego elementu do listy. Na przykład chcesz, aby w grze pojawiły się nowi obcy, chcesz uwzględnić nowe dane w wizualizacji lub chcesz dodać nowego zarejestrowanego użytkownika do budowanej witryny internetowej. Python oferuje wiele sposobów dodawania nowych danych do istniejących list.

## Umieszczanie elementu na końcu listy

Najprostszym sposobem na dodanie nowego elementu do listy jest jego *dłeganie*. Kiedy dodzasz nowy element do listy, zostaje on umieszczony na jej końcu. Wykorzystamy tę samą listę, którą mieliśmy w poprzednim przykładzie, i dołączymy na jej końcu nowy element 'ducati':

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

motorcycles.append('ducati')
print(motorcycles)
```

---

Metoda `append()` sprawia, że element 'ducati' zostaje umieszczony na końcu listy, a pozostałe elementy nie znajdują się w żaden sposób zmodyfikowane:

---

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

---

Za pomocą metody `append()` można bardzo łatwo i dynamicznie budować listy. Na przykład rozpoczynamy od pustej listy, a następnie wykorzystując serię wywołań `append()`, dodajemy kolejne elementy. Poniżej przedstawiłem przykład tego rodzaju podejścia. Na początku tworzymy pustą listę `motorcycles`, a następnie dodajemy do niej elementy `'honda'`, `'yamaha'` i `'suzuki'`.

---

```
motorcycles = []
motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

---

Otrzymana lista jest dokładnie taka sama jak ta, którą widziałeś we wcześniejszych przykładach:

---

```
['honda', 'yamaha', 'suzuki']
```

---

Listy są bardzo często budowane w taki właśnie sposób, ponieważ przed uruchomieniem programu zwykle jeszcze nie wiadomo, jakie dane będzie chciał przechowywać użytkownik. Dlatego też, aby zapewnić użytkownikowi kontrolę nad tymi danymi, rozpoczynamy od zdefiniowania pustej listy przeznaczonej na wartości wprowadzane przez użytkownika. Następnie każda dostarczana przez niego wartość zostaje umieszczona na przygotowanej wcześniej liście.

## **Wstawianie elementów na listę**

Nowy element można wstawić w dowolnie wybranym miejscu listy, używając do tego metody `insert()`. Argumentami tej metody są indeks dla nowego elementu oraz jego wartość:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

---

W tym przykładzie mamy wywołanie metody `insert()`, które powoduje wstawienie elementu `'ducati'` na początku listy. W omawianym przykładzie działanie metody `insert()` polega na zrobieniu miejsca w położeniu 0 listy, a następnie umieszczeniu w tej pozycji wartości `'ducati'`.

---

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

---

Ta operacja powoduje przesunięcie pozostałych elementów listy o jedno miejsce w prawo.

## Usuwanie elementu z listy

Często zachodzi potrzeba usunięcia z listy jednego elementu lub więcej. Przykładowo gdy graczowi uda się zestrzelić obcego, wówczas prawdopodobnie będziesz chciał tego obcego usunąć z listy aktywnych obcych. Inny przykład: gdy użytkownik zdecyduje się na zamknięciu konta w aplikacji internetowej, wtedy należy go usunąć z listy aktywnych użytkowników. Usuwać element z listy można na podstawie jego położenia, lub też wartości.

### Usunięcie elementu listy za pomocą polecenia del

Jeżeli znasz położenie elementu przeznaczonego do usunięcia z listy, możesz wykorzystać polecenie del:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

---

Kod w tym przykładzie używa polecenia del do usunięcia pierwszego elementu ('honda') z listy motocykli:

---

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

---

Za pomocą polecenia del możesz usunąć dowolny element z listy, o ile znasz jego indeks. Na przykład w poniższym fragmencie kodu usuwamy z listy drugi element ('yamaha'):

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

---

Drugi element listy został usunięty:

---

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

---

W obu przypadkach nie można już uzyskać dostępu do wartości, która została usunięta z listy za pomocą polecenia del.

## Usunięcie elementu za pomocą metody pop()

Czasami zachodzi potrzeba użycia wartości elementu już po jego usunięciu z listy. Na przykład chcemy otrzymać współrzędne  $x$  i  $y$  zestrzelonego obcego, aby dokładnie w tym miejscu wyświetlić animację eksplozji. Z kolei w przypadku aplikacji internetowej chcemy usunąć użytkownika z listy aktywnych użytkowników, a następnie umieścić go na liście nieaktywnych.

Metoda pop() powoduje usunięcie ostatniego elementu listy, ale pozwala na pracę z nim jeszcze po jego usunięciu. Pojęcie *pop* wiąże się z potraktowaniem listy jako stosu elementów, z którego można wyrzucić (*pop*) element znajdujący się na górze. W przypadku takiej analogii góra stosu odpowiada końcowi listy.

Pozbądźmy się więc motocyklu z listy motocykli:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki'] ❶
print(motorcycles)

popped_motorcycle = motorcycles.pop() ❷
print(motorcycles) ❸
print(popped_motorcycle) ❹
```

---

W wierszu ❶ definiujemy listę motocykli o nazwie `motorcycles`, którą w kolejnym kroku wyświetlamy. Następnie w wierszu ❷ pozbywamy się wartości z listy i umieszczamy ją w zmiennej `popped_motorcycle`. Po ponownym wyświetleniu listy w wierszu ❸ widzimy, że wartość została faktycznie usunięta z listy. Wyświetlamy wartość usuniętą z listy (patrz wiersz ❹), aby tym samym potwierdzić, że mamy do niej dostęp już po operacji jej usunięcia z listy.

Wygenerowane dane wyjściowe pokazują, że wartość 'suzuki' została usunięta z końca listy i jest obecnie przechowywana w zmiennej o nazwie `popped_motorcycle`:

---

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

---

Kiedy metoda `pop()` może okazać się użyteczna? Wyobraź sobie listę motocykli ułożoną w kolejności chronologicznej nabycania poszczególnych maszyn. W takim przypadku metodę `pop()` można wykorzystać do wyświetlenia komunikatu o ostatnio zakupionym motocyklu:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f'Ostatnio zakupiony przeze mnie motocykl to {last_owned.title()}.")
```

---

Dane wyjściowe powyższego fragmentu kodu to proste zdanie o najnowszym zakupionym motocyklu:

---

```
Ostatnio zakupiony przeze mnie motocykl to Suzuki.
```

---

## Usunięcie elementu z dowolnego miejsca na liście

Metodę `pop()` można wykorzystać także do usunięcia dowolnego elementu z listy. W tym celu należy indeks elementu podać w nawiasie:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']

first_owned = motorcycles.pop(0)
print(f'Mój pierwszy motocykl to {first_owned.title()}.')
```

---

Rozpoczynamy od usunięcia pierwszego elementu listy, a następnie wyświetlamy komunikat dotyczący tego motocykla. Wygenerowane dane wyjściowe to proste zdanie o pierwszym kupionym motocyku:

---

```
Mój pierwszy motocykl to Honda.
```

---

Pamiętaj, że za każdym razem, gdy używasz metody `pop()`, wskazywany przez nią element nie będzie dłużej przechowywany na liście.

Jeżeli nie jesteś pewien, czy użyć polecenia `del` czy metody `pop()`, oto prosty sposób ułatwiający podjęcie decyzji. Jeżeli chcesz usunąć element z listy i nie zamierzasz go później w żaden sposób używać, zdecyduj się na polecenie `del`. Natomiast jeśli chcesz użyć elementu po jego usunięciu, wybierz metodę `pop()`.

## Usunięcie elementu na podstawie wartości

Zdarza się, że położenie elementu przeznaczonego do usunięcia z listy jest nieznane. Jeżeli jednak znana jest jego wartość, wówczas można skorzystać z metody `remove()`.

Na przykład przyjmujemy założenie, że chcemy usunąć element 'ducati' z listy motocykli:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

motorcycles.remove('ducati')
print(motorcycles)
```

---

Tutaj metoda `remove()` nakazuje Pythonowi ustalić położenie elementu 'ducati' na liście, a następnie go usunąć:

---

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

---

Metodę `remove()` również można użyć do pracy z elementem usuniętym z listy. W poniższym fragmencie kodu usuwamy element `'ducati'` i wyświetlamy komunikat podający przyczynę usunięcia tego motocykla z listy:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati'] ❶
print(motorcycles)

too_expensive = 'ducati' ❷
motorcycles.remove(too_expensive) ❸
print(motorcycles)
print(f"\nMotocykl {too_expensive.title()} jest zbyt drogi dla mnie.") ❹
```

---

Po zdefiniowaniu listy (patrz wiersz ❶) element `'ducati'` umieszczamy w zmiennej o nazwie `too_expensive` (patrz wiersz ❷). Następnie wartość przechowywana przez tę zmienną zostaje użyta w wierszu ❸ do usunięcia elementu z listy. W prawdzie w wierszu ❹ element `'ducati'` został usunięty z listy, ale nadal jest przechowywany w zmiennej `too_expensive`, co pozwala na wyświetlenie komunikatu wyjaśniającego powód jego usunięcia z listy:

---

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

---

Motocykl Ducati jest zbyt drogi dla mnie.

---

**UWAGA** *Metoda `remove()` powoduje usunięcie tylko pierwszego wystąpienia podanej wartości. Jeżeli dana wartość może znajdować się na liście więcej niż tylko jeden raz, konieczne jest użycie pętli do ustalenia, czy wszystkie wystąpienia tej wartości zostały usunięte z listy. W rozdziale 7. dowiesz się, jak można to zrobić.*

## ZRÓB TO SAM

Poniższe ćwiczenia są nieco bardziej skomplikowane niż te przedstawione w rozdziale 2., ale dają możliwość użycia list na wszystkie omówione dotąd sposoby.

**3.4. Lista gości.** Jeżeli mógłbyś zaprosić kogokolwiek na obiad, żywiącego lub nieżyjącego, to kogo byś zaprosił? Utwórz listę zawierającą przynajmniej trzy osoby, które chciałbyś zaprosić na obiad. Następnie wykorzystaj tę listę do wyświetlenia dla każdej z tych osób komunikatu zapraszającego ją na obiad.

**3.5. Zmiana listy gości.** Dowiedziałeś się, że jedna z zaproszonych osób nie może przyjść na obiad. Konieczne jest więc wysłanie następnych zaproszeń. Zastanów się, kogo w takim razie jeszcze zaprosisz.

- Pracę rozpocznij od programu utworzonego w ćwiczeniu 3.4. Na jego końcu umieść polecenie `print()` wyświetlające komunikat z informacją, który z zaproszonych gości nie może przyjść.
- Zmodyfikuj listę i dane gościa, który nie będzie mógł przybyć na obiad, zastąp danymi nowej zaproszonej osoby.
- Wyświetl drugi zestaw komunikatów z zaproszeniem, po jednym komunikacie dla każdej osoby znajdującej się na liście.

**3.6. Więcej gości.** Znalazłeś większy stół, co oznacza więcej miejsca dla gości. Zastanów się więc nad jeszcze trzema osobami, które mógłbyś zaprosić na obiad.

- Pracę rozpocznij od programu utworzonego w ćwiczeniach 3.4 i 3.5. Na jego końcu umieść polecenie `print()` wyświetlające komunikat o znalezieniu większego stołu.
- Za pomocą metody `insert()` dodaj nowego gościa na początku listy.
- Za pomocą metody `insert()` dodaj nowego gościa w środku listy.
- Za pomocą metody `append()` dodaj nowego gościa na końcu listy.
- Wyświetl nowy zestaw komunikatów z zaproszeniem, po jednym komunikacie dla każdej osoby znajdującej się na liście.

**3.7. Kurcząca się lista gości.** Okazało się, że większy stół nie zostanie dostarczony na czas i dlatego masz miejsce dla jedynie dwóch gości.

- Pracę rozpocznij od programu utworzonego w ćwiczeniu 3.6. Dodaj nowy wiersz wyświetlający komunikat, że na obiad możesz zaprosić tylko dwie osoby.
- Za pomocą metody `pop()` usuwaj po jednym gościu z listy, aż zostaną na niej tylko dwie osoby. Po usunięciu każdej osoby wyświetlaj przeznaczony dla niej komunikat z przeprosinami za brak możliwości zaproszenia jej na obiad.
- Oba osobom pozostałym na liście wyświetl spersonalizowany komunikat z zaproszeniem na obiad.
- Użyj polecenia `del` do usunięcia dwóch ostatnich osób z listy, która w ten sposób powinna stać się pusta. Na koniec wyświetl listę, aby upewnić się, że faktycznie jest pusta.

## Organizacja listy

Często listy są tworzone w nieprzewidywalnej kolejności, ponieważ nie zawsze można kontrolować kolejność, w jakiej użytkownicy będą wprowadzać dane. Z jednej strony jest to nieuniknione w większości przypadków, ale z drugiej —

zwykle chcesz przedstawiać informacje w określonej kolejności. Czasami zachodzi potrzeba zachowania pierwotnej kolejności listy, natomiast w innych sytuacjach należy ją zmienić. Python oferuje wiele różnych sposobów organizacji listy, w zależności od sytuacji.

## Trwałe sortowanie listy za pomocą metody sort()

Sortowanie listy za pomocą metody `sort()` Pythona jest dość łatwym zadaniem. Wyobraź sobie, że mamy listę producentów samochodów i chcemy zmienić ich kolejność na alfabetyczną. Aby zachować prostotę zadania zakładamy, że wszystkie wartości na liście zostały zapisane małymi literami.

*Plik cars.py:*

---

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

---

Użyta w kodzie metoda `sort()` powoduje trwałą zmianę kolejności elementów listy. Marki samochodów są teraz ułożone w kolejności alfabetycznej i nie ma możliwości powrotu do pierwotnej kolejności elementów na liście:

---

```
['audi', 'bmw', 'subaru', 'toyota']
```

---

Listę można posortować także w odwrotnej kolejności alfabetycznej, co wymaga przekazania metodzie `sort()` argumentu `reverse=True`. W poniższym fragmencie kodu przedstawiłem przykład sortowania listy w odwrotnej kolejności alfabetycznej:

---

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

---

Także w tym przypadku kolejność elementów listy została trwale zmieniona:

---

```
['toyota', 'subaru', 'bmw', 'audi']
```

---

## Tymczasowe sortowanie listy za pomocą funkcji `sorted()`

W celu zachowania pierwotnej kolejności elementów na liście, ale wyświetlenia ich jako elementów posortowanych, można użyć funkcji o nazwie `sorted()`. Ta funkcja pozwala na wyświetlenie elementów listy we wskazanej kolejności, choć nie wpływa na rzeczywistą ich kolejność na tej liście.

Wypróbowajmy działanie funkcji sorted() na utworzonej wcześniej liście samochodów:

---

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Oto lista początkowa:") ❶
print(cars)

print("\nOto lista posortowana:") ❷
print(sorted(cars))

print("\nOto ponownie lista początkowa:") ❸
print(cars)
```

---

Polecenie w wierszu ❶ powoduje wyświetlenie listy z początkową kolejnością elementów, natomiast polecenie w wierszu ❷ wyświetla tę samą listę, ale już z posortowanymi elementami. Następnie ponownie wyświetlamy listę początkową z pierwotną kolejnością elementów, aby pokazać, że działanie funkcji sorted() jest jedynie tymczasowe (patrz wiersz ❸):

---

```
Oto lista początkowa:
['bmw', 'audi', 'toyota', 'subaru']

Oto lista posortowana:
['audi', 'bmw', 'subaru', 'toyota']

Oto ponownie lista początkowa: ❶
['bmw', 'audi', 'toyota', 'subaru']
```

---

Zwróć uwagę, że mimo użycia przez Ciebie funkcji sorted() lista nadal zachowała swoją pierwotną kolejność (patrz wiersz ❶). Warto w tym miejscu dodać, że funkcja sorted() również akceptuje argument reverse=True, który okazuje się przydatny, jeśli chcesz wyświetlić listę elementów w odwrotnej kolejności alfabetycznej.

**UWAGA** Sortowanie listy w kolejności alfabetycznej jest nieco bardziej skomplikowane, gdy nie wszystkie wartości są zapisane małymi literami. Istnieje kilka sposobów na interpretację wielkich liter w trakcie sortowania, a określenie dokładnie interesującej nas kolejności może być znacznie bardziej skomplikowane niż tego oczekujemy na tym etapie poznawania Pythona. Jednak większość podejść dotyczących sortowania będzie budowanych bezpośrednio na koncepcjach przedstawionych w tym rozdziale.

## Wyświetlanie listy w odwrotnej kolejności alfabetycznej

W celu odwrócenia pierwotnej kolejności listy można użyć metody `reverse()`. Jeżeli pierwotna lista zawierała samochody ułożone w chronologicznej kolejności ich kupowania, wtedy bardzo łatwo można zmienić tę kolejność na odwrotnie chronologiczną:

---

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

---

Zwrć uwagę na to, że metoda `reverse()` nie przeprowadza sortowania w odwrotnej kolejności chronologicznej, a po prostu odwraca kolejność listy:

---

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

---

Metoda `reverse()` trwale zmienia kolejność listy, ale istnieje możliwość przywrócenia pierwotnej kolejności elementów na liście po prostu przez ponowne wykonanie metody `reverse()`.

## Określenie wielkości listy

Wielkość listy można dość szybko określić za pomocą funkcji `len()`. Przedstawiona poniżej lista zawiera cztery elementy, stąd jej wielkość wynosi 4:

---

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

---

Funkcja `len()` może okazać się użyteczna, gdy na przykład zajdzie potrzeba ustalenia liczby obcych pozostałych do zneutralizowania w grze, ustalenia ilości danych, jakimi trzeba zarządzać w wizualizacji, a także gdy zajdzie potrzeba ustalenia liczby zarejestrowanych użytkowników witryny internetowej.

**UWAGA** *Elementy listy są w Pythonie liczone od jednego, więc podczas ustalania wielkości listy nie powinny występować żadnego rodzaju błędy związane z przesunięciem o jeden.*

## ZRÓB TO SAM

**3.8. Zwiedzaj świat.** Pomyśl o pięciu miejscach na świecie, które chciałbyś odwiedzić.

- Wszystkie miejsca umieść na liście i upewnij się, że nie jest ona ułożona alfabetycznie.
- Wyświetl listę w jej pierwotnej kolejności. Nie przejmuj się eleganckim wyświetleniem listy, a po prostu wyświetl ją jako zwykłą listę Pythona.
- Za pomocą funkcji `sorted()` wyświetl listę w kolejności alfabetycznej bez modyfikacji rzeczywistej listy.
- Ponownie wyświetl listę początkową, aby pokazać, że nie została zmodyfikowana.
- Za pomocą funkcji `sorted()` wyświetl listę w odwrotnej kolejności alfabetycznej bez modyfikacji rzeczywistej listy.
- Ponownie wyświetl listę początkową, aby pokazać, że nie została zmodyfikowana.
- Za pomocą metody `reverse()` zmień kolejność listy, a następnie wyświetl ją, aby potwierdzić zmianę kolejności.
- Ponownie wykorzystaj metodę `reverse()` do zmiany kolejności listy. Wyświetl ją, aby pokazać, że powróciła do pierwotnej kolejności.
- Za pomocą metody `sort()` zmień kolejność listy na alfabetyczną, a następnie wyświetl ją, aby potwierdzić zmianę kolejności.
- Za pomocą metody `sort()` zmień kolejność listy na odwrotnie alfabetyczną, a następnie wyświetl ją, aby potwierdzić zmianę kolejności.

**3.9. Goście na obiad.** Pracę rozpocznij od jednego z programów utworzonych w ćwiczeniach od 3.4 do 3.7. Za pomocą funkcji `len()` wyświetl komunikat wskazujący liczbę osób, które zostały zaproszone na obiad.

**3.10. Każda funkcja.** Zastanów się, jakie informacje mógłbyś umieścić na liście. Na przykład utwórz listę gór, rzek, państw, miast, języków lub czegokolwiek innego. Przygotuj program, który będzie tworzył listę przechowującą te informacje i używał co najmniej jeden raz każdej funkcji wprowadzonej w tym rozdziale.

## Unikanie błędów indeksu podczas pracy z listą

Jeden rodzaj błędu dość często występuje podczas pracy z listą. Przyjmujemy założenie, że mamy listę składającą się z trzech elementów i próbujemy uzyskać dostęp do czwartego:

## Plik motorcycles.py:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

---

W takiej sytuacji wygenerowany zostanie *błęd indeksu*:

---

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[3])
               ^^^
IndexError: list index out of range
```

---

Python próbuje przekazać element znajdujący się na liście w położeniu o indeksie 3. Jednak kiedy przeszukał listę motorcycles, okazało się, że lista ta nie zawiera żadnego elementu o indeksie 3. Ze względu na przesunięcie o jeden numeru indeksu i elementu na liście tego rodzaju błąd występuje dość często. Programiści często myślą, że trzeci element listy ma indeks 3, ponieważ zaczynają liczyć od jednego. Jednak w Pythonie trzeci element listy ma indeks 2, ponieważ numeracja indeksów rozpoczyna się od zera.

Błąd indeksu oznacza, że Python nie znajduje elementu o żądanym indeksie. Jeżeli tego rodzaju błąd wystąpi w programie, spróbuj zmienić (zmniejszyć) o jeden wartość indeksu. Następnie ponownie uruchom program i zobacz, czy otrzymasz prawidłowy wynik.

Warto w tym miejscu pamiętać, że jeśli chcesz uzyskać dostęp do ostatniego elementu listy, możesz użyć indeksu -1. To zawsze działa, nawet jeśli lista uległa zmianie od ostatniej operacji na niej przeprowadzonej:

---

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

---

Indeks -1 zawsze zwraca ostatni element listy, którym w omawianym przykładzie jest 'suzuki':

---

```
'suzuki'
```

---

Jedyna sytuacja, w której użycie indeksu -1 zakończy się błędem, to próba pobrania ostatniego elementu z pustej listy:

---

```
motorcycles = []
print(motorcycles[-1])
```

---

Ponieważ lista `motorcycles` nie zawiera żadnych elementów, Python zgłasza błąd indeksu:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[-1])
                ^~~~^~~~
IndexError: list index out of range
```

Jeżeli wystąpi błąd indeksu i nie będziesz umiał sobie z nim poradzić, spróbuj wyświetlić zawartość listy lub jej wielkość. Lista może w rzeczywistości przedstawiać się zupełnie inaczej, niż sądzisz, zwłaszcza jeśli powstaje dynamicznie w trakcie działania programu. Spójrz na rzeczywistą listę lub sprawdź liczbę znajdujących się na niej elementów — może to pomóc w rozwiązyaniu tego rodzaju błędów logicznych.

### ZRÓB TO SAM

**3.11. Celowy błąd.** Jeżeli jeszcze w żadnym z tworzonych programów nie spotkałeś się z błędem indeksu, spróbuj go teraz wywołać. W dowolnym programie tak zmień wartość indeksu, aby nastąpiło zgłoszenie błędu indeksu. Zanim zamkniesz plik programu, upewnij się, że usunąłeś błąd.

## Podsumowanie

W tym rozdziale dowiedziałeś się, czym jest lista oraz jak można pracować z jej poszczególnymi elementami. Zobaczyłeś, jak zdefiniować listę, a także jak dodać do niej elementy i je z niej usuwać. Zajmowałeś się trwałym i tymczasowym sortowaniem zawartości listy. Nauczyłeś się również ustalać wielkość listy oraz unikać błędów indeksu podczas pracy z listą.

W rozdziale 4. zobaczyłeś, jak można znacznie efektywniej pracować z elementami listy. Dzięki wykorzystaniu pętli uzyskuje się dostęp do każdego elementu listy za pomocą zaledwie kilku wierszy kodu. W ten sposób można efektywnie pracować z listami zawierającymi nawet tysiące lub miliony elementów.

# 4

## Praca z listą



W rozdziale 3. DOWIEDZIAŁEŚ SIĘ, JAK UTWORZYĆ PROSTĄ LISTĘ ORAZ PRACOWAĆ Z JEJ POSZCZEGÓLNYMI ELEMENTAMI. W TYM ROZDZIALE ZOBACZYSZ, JAK ZA POMOCĄ **PĘTLI** MOŻESZ PRZEPROWADZIĆ ITERACJĘ przez całą listę (niezależnie od jej wielkości), używając do tego zaledwie kilku wierszy kodu. *Pętla* umożliwia podjęcie tej samej akcji lub zbioru akcji na każdym elemencie listy. W ten sposób zyskujesz możliwość efektywnej pracy z listami o dowolnej wielkości, również z tymi zawierającymi tysiące lub miliony elementów.

### Iteracja przez całą listę

Bardzo często zachodzi konieczność przeprowadzenia iteracji przez wszystkie elementy listy i wykonania tej samej operacji na każdym z nich. Może na przykład wystąpić potrzeba przesunięcia o tę samą odległość wszystkich elementów wyświetlanych na ekranie. Ewentualnie na wszystkich elementach listy zawierającej dane liczbowe chcesz przeprowadzić dokładnie te same operacje statystyczne. Jeszcze inną możliwość to konieczność wyświetlenia każdego nagłówka z listy artykułów w witrynie internetowej. Kiedy chcesz przeprowadzić tę samą akcję względem każdego elementu listy, możesz wykorzystać oferowaną przez Pythona pętlę `for`.

Przyjmujemy założenie, że mamy listę imion magików i chcemy wyświetlić je wszystkie. Wprawdzie można to zrobić, pobierając jednorazowo po jednym imieniu, ale takie podejście rodzi wiele problemów. Przede wszystkim będzie uciążliwe w przypadku długiej listy imion. Ponadto kod odpowiedzialny za ich wyświetlanie

trzeba będzie modyfikować po każdej zmianie wprowadzonej na liście magików. Dzięki użyciu pętli `for` można uniknąć obu wymienionych problemów i pozwoić Pythonowi na wewnętrzne zajęcie się nimi.

Wykorzystamy teraz pętlę `for` do wyświetlenia imion wszystkich magików umieszczonych na liście.

*Plik `magicians.py`:*

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(magician)
```

---

Rozpoczynamy od zdefiniowania listy, podobnie jak to robiliśmy już w rozdziale 3. Następnie definiujemy pętlę `for`. To polecenie nakazuje Pythonowi pobranie imienia z listy `magicians` oraz jego umieszczenie w zmiennej `magician`. Ostatnim krokiem jest wyświetlenie imienia przechowywanego w zmiennej `magician`. Teraz Python ponownie wykonuje dwa ostatnie polecenia, jednokrotnie dla każdego imienia na liście. Powyższy kod można odczytać następująco: „Wyświetl imię każdego magika wymienionego na liście magików”. Wygenerowane dane wyjściowe to prosta lista imion:

---

```
alicja
dawid
karolina
```

---

## Dokładniejsza analiza pętli

Koncepcja pętli jest ważna, ponieważ to jeden z najczęściej stosowanych przez komputer sposobów automatyzacji powtarzających się zadań. Przykładowo w prostej pętli, takiej jak ta użyta w `magicians.py`, Python na początku odczytuje pierwszy wiersz pętli:

---

```
for magician in magicians:
```

---

Powyższe polecenie nakazuje Pythonowi pobranie pierwszej wartości z listy `magicians` oraz umieszczenie jej w zmiennej `magician`. Pierwsza wartość to '`alicja`'. Następnie Python przechodzi do kolejnego wiersza kodu:

---

```
print(magician)
```

---

To polecenie powoduje wyświetlenie bieżącej wartości zmiennej `magician`, którą nadal jest '`alicja`'. Ponieważ lista zawiera więcej wartości, Python powraca do pierwszego wiersza pętli:

---

```
for magician in magicians:
```

---

Python pobiera następne imię z listy (tutaj 'dawid') i umieszcza je w zmiennej `magician`. Teraz Python przechodzi do kolejnego wiersza pętli:

---

```
print(magician)
```

---

Ponownie wyświetlna jest bieżąca wartość zmiennej `magician`, którą teraz jest 'dawid'. Cała pętla jest wykonywana raz jeszcze dla ostatniej wartości na liście ('karolina'). Ponieważ lista nie zawiera więcej wartości, Python przechodzi do wykonania następnego wiersza kodu w programie. W omawianym przykładzie nie ma żadnych poleceń po pętli `for`, więc następuje po prostu zakończenie działania programu.

Jeżeli używasz pętli po raz pierwszy, to zapamiętaj, że zdefiniowany w niej zbiór poleceń będzie wykonany jednokrotnie dla każdego elementu listy niezależnie od tego, ile elementów znajduje się na liście. Jeżeli lista zawiera milion elementów, Python powtórzy te kroki milion razy i zwykle zrobi to bardzo szybko.

Podczas tworzenia własnych pętli `for` warto pamiętać jeszcze o jednej kwestii. Wprawdzie można wybrać dowolną nazwę dla zmiennej tymczasowej przechowującej poszczególne wartości listy, ale rozsądne będzie użycie czytelnej nazwy przedstawiającej pojedynczy element listy. Przykładowo poniżej przedstawiłem dobry początek dla pętli `for` przeprowadzających iteracje przez listy kotów, psów oraz ogólną listę elementów:

---

```
for cat in cats:  
for dog in dogs:  
for item in list_of_items:
```

---

Takie konwencje nazewnicze pomogą Ci podążać za akcjami wykonywanymi dla poszczególnych elementów wewnętrz pętli `for`. Użycie liczby pojedynczej i mnogiej będzie pomocne w ustaleniu, który fragment kodu działa z pojedynczym elementem listy, a który został przeznaczony dla listy jako całości.

## Wykonanie większej liczby zadań w pętli `for`

W pętli `for` z każdym elementem można zrobić praktycznie wszystko. Rozbudujemy poprzedni przykład w ten sposób, że dodamy komunikat dla każdego magika, zawierający podziękowanie za wykonanie wspaniałej sztuczki:

*Plik `magicians.py`:*

---

```
magicians = ['alicja', 'dawid', 'karolina']  
for magician in magicians:  
    print(f"{magician.title()}, to była doskonała sztuczka!")
```

---

Jedyna różnica w stosunku do poprzedniego kodu polega na utworzeniu komunikatu przeznaczonego dla magika i rozpoczynającego się od jego imienia. W trakcie pierwszej iteracji pętli wartością zmiennej `magician` jest 'alicja'. Dlatego też Python rozpoczyna pierwszy komunikat od imienia 'Alicja'. W trakcie drugiej iteracji pętli na początku komunikatu mamy imię 'Dawid', natomiast w trzeciej iteracji pętli komunikat rozpoczyna się od imienia 'Karolina'.

Wygenerowane dane wyjściowe to spersonalizowany komunikat dla każdego magika wymienionego na liście:

---

```
Alicja, to była doskonała sztuczka!
Dawid, to była doskonała sztuczka!
Karolina, to była doskonała sztuczka!
```

---

W pętli `for` można umieścić dowolną liczbę wierszy kodu. Pamiętaj, że każdy wcięty wiersz pod `magician` in `magicians`: jest uznawany za *kod we wnętrzu pętli*, więc wszystkie wcięte wiersze są wykonywane jednokrotnie dla każdego elementu znajdującego się na liście. Tym samym względem poszczególnych wartości na liście można wykonać dowolną liczbę operacji.

Do generowanego komunikatu dodamy teraz drugi wiersz informujący magika o tym, jak niecierpliwie czekamy na kolejny jego występ:

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(f"{magician.title()}, to była doskonała sztuczka!")
    print(f"Nie mogę się doczekać Twojej kolejnej sztuczki,
{magician.title()}.\\n")
```

---

Ponieważ oba wiersze zawierające wywołania `print()` są wcięte, każdy z nich będzie wykonany jednokrotnie dla wszystkich magików wymienionych na liście. Znak nowego wiersza ("\\n") umieszczony w drugim wywołaniu `print()` powoduje wstawienie pustego wiersza po każdej iteracji pętli. W ten sposób tworzymy zbiór wiadomości, w którym komunikaty dla poszczególnych osób wymienionych na liście są elegancko pogrupowane:

---

```
Alicja, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Alicja.
```

```
Dawid, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Dawid.
```

```
Karolina, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Karolina.
```

---

W pętli `for` możesz umieścić dowolną liczbę wierszy kodu. W praktyce często się przekonasz, że za pomocą pętli `for` można efektywnie wykonywać wiele różnych operacji dla poszczególnych elementów listy.

## Wykonywanie operacji po pętli `for`

Co się dzieje po zakończeniu wykonywania pętli `for`? Zwykle chcemy podsumować blok danych wyjściowych lub przejść do innego bloku operacji, które mają być przeprowadzone przez program.

Wszystkie znajdujące się po pętli `for` wiersze kodu, które nie są wcięte, zostaną wykonane już tylko jeden raz, bez powtórzenia. Utworzmy teraz komunikat dla grupy magików jako całości, w którym podziękujemy im za doskonały występ. Aby wyświetlić ten komunikat po wszystkich wiadomościach przeznaczonych dla poszczególnych magików, umieszczać wywołanie `print()` po pętli `for` i nie stosujemy wcięcia dla tego wiersza kodu:

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(f"{magician.title()}, to była doskonała sztuczka!")
    print(f"Nie mogę się doczekać Twojej kolejnej sztuczki,
{magician.title()}.\n")

print("Dziękuję wszystkim. To był naprawdę wspaniały występ!")
```

---

Pierwsze dwa wywołania `print()` w powyższym kodzie źródłowym będą powtarzane dla każdego magika wymienionego na liście, o czym się przekonałeś już wcześniej. Jednak wiersz nie jest wcięty i dlatego zostanie wykonany tylko raz:

---

```
Alicja, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Alicja.
```

---

```
Dawid, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Dawid.
```

```
Karolina, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Karolina.
```

---

```
Dziękuję wszystkim. To był naprawdę wspaniały występ!
```

---

Podczas przetwarzania danych za pomocą pętli `for` zobacysz, że to jest doskonały sposób na podsumowanie operacji przeprowadzonej na całym zbiorze danych. Pętlę `for` możesz na przykład wykorzystać do zainicjalizowania gry, przeprowadzając iterację przez listę postaci i wyświetlając je wszystkie na ekranie. Następnie po pętli umieścisz pozbawiony wcięć blok kodu odpowiedzialny za wyświetlenie przycisku, który umożliwi rozpoczęcie gry, gdy wszystkie postacie zostaną już umieszczone na ekranie.

# Unikanie błędów związanych z wcięciami

Python stosuje wcięcia w celu ustalenia, czy dany wiersz kodu jest powiązany z wierszem znajdującym się powyżej. We wcześniejszych przykładach wiersze wyświetlające komunikaty dla poszczególnych magików były uznawane za część pętli `for`, ponieważ zostały wcięte. Dzięki wcięciom kod Pythona jest niezwykle łatwy do odczytania. W zasadzie wykorzystuje białe znaki do wymuszenia eleganckiego formatowania kodu i stosowania przejrzystej struktury wizualnej. W dłuższych programach pisanych w Pythonie będziesz mógł zauważać bloki kodu wraz z wcięciami na kilku różnych poziomach. Tego rodzaju wcięcia pomagają w określeniu ogólnej organizacji kodu źródłowego programu.

Gdy rozpocznesz tworzenie kodu opierającego się na stosowaniu prawidłowych wcięć, spotkasz się z kilkoma najczęściej występującymi *błędami wcięć*. Zdarza się na przykład, że programiści stosują wcięcia dla bloków kodu, które nie powinny być wcięte, lub na odwrót — zapominają o wcięciach tam, gdzie są one niezbędne. Przeanalizowanie przykładów tego rodzaju błędów pomoże uniknąć ich w przyszłości oraz usunąć je, gdy już wystąpią w tworzonych przez Ciebie programach.

Przechodzimy więc teraz do kilku najczęściej występujących błędów związanych z wcięciami.

## Brak wcięcia

Wiersz kodu tuż po pętli `for` zawsze powinien być wcięty. Jeżeli o tym zapomnisz, Python na pewno Ci przypomni.

Plik `magicians.py`:

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(magician) ❶
```

---

Polecenie w wierszu ❶ powinno być wcięte, ale nie jest. Kiedy Python oczekuje wciętego bloku kodu i go nie znajduje, wówczas wyświetla odpowiedni komunikat i wskazuje problematyczny wiersz:

---

```
File "magicians.py", line 3
    print(magician)
          ^
IndentationError: expected an indented block after 'for' statement on line 2
```

---

Tego rodzaju błąd można zwykle usunąć przez wcięcie wiersza lub wierszy znajdujących się po poleceniu `for`.

## Brak wcięcia dodatkowych wierszy

Czasami pętla działa bez jakiegokolwiek błędu, ale nie powoduje wygenerowania oczekiwanych danych wyjściowych. Tego rodzaju sytuacja zdarza się, gdy w bloku pętli chcesz wykonać kilka zadań, ale zapomnisz o wcięciu pewnych wierszy.

Zobaczmy, co się stanie, gdy na przykład zapomnimy o wcięciu w bloku pętli drugiego wiersza odpowiedzialnego za wyświetlenie magikowi komunikatu o oczekiwaniu na jego kolejną sztuczkę:

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(f"{magician.title()}, to była doskonała sztuczka!")
print(f"Nie mogę się doczekać Twojej kolejnej sztuczki, {magician.title()}.\\n") ❶
```

---

Wywołanie `print()` w wierszu ❶ powinno być wcięte. Jednak skoro po poleceniu `for` Python znalazł jeden wcięty wiersz kodu, nie zgłasza błędu. Pierwsze wywołanie `print()` jest wykonywane jednokrotnie dla każdego magika znajdującej się na liście, ponieważ zostało wcięte. Z kolei drugie nie jest wcięte i zostanie wykonane tylko jednokrotnie dopiero po zakończeniu działania pętli. Ostatnim magikiem na liście jest 'Karolina', więc jedynie dla niej zostanie wyświetlony komunikat z informacją o oczekiwaniu na jej kolejną sztuczkę:

---

```
Alicja, to była doskonała sztuczka!
Dawid, to była doskonała sztuczka!
Karolina, to była doskonała sztuczka!
Nie mogę się doczekać Twojej kolejnej sztuczki, Karolina.
```

---

Mamy tutaj do czynienia z *błądem logicznym*. Składnia kodu Pythona jest prawidłowa, ale kod nie generuje oczekiwanych danych wyjściowych z powodu istnienia problemu w jego logice. Jeżeli oczekujesz wykonania określonej akcji dla każdego elementu listy, a obserwujesz tylko jednorazowe jej wykonanie, sprawdź, czy po prostu nie trzeba zastosować wcięcia jeszcze dla co najmniej jednego wiersza.

## Niepotrzebne wcięcie

Jeżeli zastosujesz wcięcie wiersza tam, gdzie jest ono niepotrzebne, Python poinformuje Cię o tym.

*Plik hello\_world.py:*

---

```
message = "Witaj, świecie Pythona!"
    print(message)
```

---

Nie ma potrzeby wcięcia polecenia `print()`, ponieważ *nie należy* on do wiersza znajdującego się powyżej. Dlatego też Python zgłasza błąd:

---

```
File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

---

Błędów nieoczekiwanych wcięć można unikać przez stosowanie wcięć tylko wtedy, gdy istnieją ku temu ważne powody. W tworzonych teraz przez Ciebie programach wcięcia powinny pojawiać się jedynie dla akcji, które mają być wykonywane dla każdego elementu przetwarzanego przez pętlę `for`.

## Niepotrzebne wcięcie po pętli

Jeżeli zastosujesz przypadkowe wcięcie wiersza kodu, który powinien zostać wykonany po zakończeniu działania pętli, kod ten zostanie wywołany dla każdego elementu listy przetwarzanej przez pętlę. Czasami to spowoduje zgłoszenie błędu przez Pythona, choć najczęściej będziesz mieć do czynienia po prostu z błędem logicznym.

Zobaczmy na przykład, co się stanie, gdy przypadkowo zastosujemy wcięcie dla wiersza zawierającego podziękowanie dla wszystkich magików:

---

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians:
    print(f"{magician.title()}, to była doskonała sztuczka!")
    print(f"Nie mogę się doczekać Twojej kolejnej sztuczki,
{magician.title()}.\\n")
    print("Dziękuję wszystkim. To był naprawdę wspaniały występ!") ❶
```

---

Ponieważ wiersz ❶ został wcięty, będzie wykonywany dla każdego magika wymienionego na liście, o czym możesz się przekonać, patrząc na poniższe dane wyjściowe:

---

Alicja, to była doskonała sztuczka!  
Nie mogę się doczekać Twojej kolejnej sztuczki, Alicja.

Dziękuję wszystkim. To był naprawdę wspaniały występ! □  
Dawid, to była doskonała sztuczka!  
Nie mogę się doczekać Twojej kolejnej sztuczki, Dawid.

Dziękuję wszystkim. To był naprawdę wspaniały występ! □  
Karolina, to była doskonała sztuczka!  
Nie mogę się doczekać Twojej kolejnej sztuczki, Karolina.

Dziękuję wszystkim. To był naprawdę wspaniały występ! □

---

To jest inny rodzaj błędu logicznego, podobnego do przedstawionego wcześniej w sekcji „Brak wcięcia dodatkowych wierszy”. Ponieważ Python nie wie, co zamierzasz osiągnąć w kodzie, wykonuje cały kod, który charakteryzuje się prawidłową składnią. Jeżeli akcja jest powtórzona wielokrotnie, choć powinna być wykonana tylko raz, sprawdź, czy przypadkiem nie zastosowałeś wcięcia w miejscu, w którym go nie powinno być.

## Brak dwukropka

Dwukropka na końcu polecenia `for` nakazuje Pythonowi zinterpretować następny wiersz jako początek pętli:

```
magicians = ['alicja', 'dawid', 'karolina']
for magician in magicians ❶
    print(magician)
```

Jeżeli przypadkowo zapomnisz o tym dwukropku, jak pokazałem w wierszu ❶, skutkiem będzie wygenerowanie błędu składni, ponieważ Python nie wie, co zamierzasz osiągnąć.

```
File "magicians.py", line 2
    for magician in magicians
               ^
SyntaxError: expected ':'
```

Python nie wie, czy po prostu zapomniałeś dodać dwukropka, czy zamierzałeś utworzyć dodatkowy kod w celu zdefiniowania znacznie bardziej złożonej pętli. Jeżeli interpreter jest w stanie określić potencjalne rozwiązanie, zasugeruje je, np. poprzez umieszczenie dwukropka na końcu wiersza, jak ma to miejsce w omawianym przykładzie (komunikat `expected ':'`). Część błędów jest łatwa do usunięcia, dzięki sugestiom Pythona umieszczonym na stosie wywołań. Z kolei inne znacznie trudniej usunąć, nawet jeśli ostateczna poprawka dotyczy tylko pojedynczego znaku. Nie irytuj się, gdy wprowadzenie drobnej poprawki wymagało dużo czasu poświęconego na odszukanie błędu — nie tylko Ty miałeś takie problemy.

## ZRÓB TO SAM

**4.1. Pizza.** Pomyśl o przynajmniej trzech rodzajach pizzy. Umieść te nazwy na liście, a następnie wyświetl je za pomocą pętli `for`.

- Zmodyfikuj pętlę `for` w taki sposób, aby zamiast samej nazwy pizzy wyświetlała zdanie wykorzystujące tę nazwę. Dla każdego rodzaju pizzy powinieneś mieć jeden wiersz danych wyjściowych zawierający proste zdanie w stylu: „Lubię pizzę pepperoni”.

- Poza blokiem pętli, na końcu programu dodaj zdanie informujące o tym, jak bardzo lubisz pizzę. Wygenerowane dane wyjściowe powinny więc składać się z co najmniej trzech zdań o Twoich ulubionych rodzajach pizzy oraz jednego zdania dodatkowego w stylu: „Naprawdę uwielbiam pizzę!“.

**4.2. Zwierzęta.** Spróbuj znaleźć trzy różne zwierzęta charakteryzujące się podobnymi cechami. Umieść ich nazwy na liście, a następnie wyświetl je za pomocą pętli for.

- Zmodyfikuj program w taki sposób, aby wyświetlał zdanie dotyczące danego zwierzęcia, na przykład „Pies jest prawdziwym przyjacielem człowieka“.
- Na końcu programu umieść zdanie wskazujące na jakieś podobieństwo między wymienionymi wcześniej zwierzętami. To może być zdanie w stylu: „Wszystkie wymienione powyżej zwierzęta są wspaniałe!“.

## Tworzenie list liczbowych

Istnieje wiele powodów przechowywania zbiorów liczb. Na przykład konieczność monitorowania położenia każdej postaci w grze lub najlepszych wyników uzyskanych przez gracza. W przypadku wizualizacji danych niemal zawsze mamy do czynienia ze zbiorami liczb, takimi jak temperatura, odległości, wielkość populacji czy współrzędne geograficzne.

Lista idealnie nadaje się do przechowywania zbiorów liczb, a Python oferuje wiele narzędzi pomagających wydajnie pracować z listami zawierającymi liczby. Gdy tylko nauczysz się efektywnie używać tych narzędzi, tworzony przez Ciebie kod będzie doskonale przygotowany do pracy nawet z listami zawierającymi miliony elementów.

### Użycie funkcji range()

Funkcja range() ułatwia generowanie serii liczb. Przykładowo za jej pomocą można w pokazany poniżej sposób wygenerować serię liczb.

Plik first\_numbers.py:

---

```
for value in range(1, 5):
    print(value)
```

---

Wprawdzie powyższy kod może sugerować wyświetlenie liczb od 1 do 5, ale tak naprawdę nie wyświetli liczby 5:

---

```
1
2
3
4
```

---

W omawianym przykładzie funkcja `range()` wyświetla jedynie liczby od 1 do 4. To jest następny efekt wspomnianego wcześniej przesunięcia o jeden, z którym będziesz się spotykać w językach programowania. Funkcja `range()` nakazuje Pythonowi rozpocząć odliczanie od pierwszej podanej wartości i zatrzymać się po dotarciu do drugiej. Ponieważ odliczanie zatrzymuje się na drugiej wartości, dane wyjściowe nigdy nie będą wyświetlały tej wartości końcowej, którą w omawianym przykładzie jest 5.

Jeżeli chcesz wyświetlić liczby od 1 do 5, musisz użyć wywołania `range(1, 6)`:

---

```
for value in range(1, 6):
    print(value)
```

---

Tym razem dane wyjściowe zawierają liczby od 1 do 5:

---

```
1
2
3
4
5
```

---

Jeżeli używając funkcji `range()`, otrzymujesz dane wyjściowe inne niż oczekiwane, spróbuj zmienić wartość końcową o 1.

Funkcji `range()` można przekazać tylko jeden argument, wówczas sekwencja liczb będzie rozpoczynała się od 0. Na przykład wywołanie `range(6)` zwróci liczby od 0 do 5.

## Użycie funkcji `range()` do utworzenia listy liczb

Jeżeli chcesz utworzyć listę liczb, wynik wywołania `range()` możesz bezpośrednio skonwertować na listę za pomocą funkcji `list()`. Kiedy opakujesz wywołanie `range()` funkcją `list()`, wygenerowane dane wyjściowe będą listą liczb.

W przykładzie przedstawionym w poprzedniej sekcji po prostu wyświetliśmy serię liczb. Z kolei funkcja `list()` pozwala skonwertować ten sam zbiór liczb na listę:

---

```
numbers = list(range(1, 6))
print(numbers)
```

---

Oto wynik działania powyższego fragmentu kodu.

---

```
[1, 2, 3, 4, 5]
```

---

Funkcję `range()` można nawet wykorzystać, aby nakazać Pythonowi pominięcie liczb w podanym zakresie. Po przekazaniu trzeciego argumentu funkcji `range()` każda kolejna generowana liczba będzie zwiększana o wartość tego argumentu.

Przykładowo poniżej pokazalem, jak można utworzyć listę liczb parzystych z zakresu od 1 do 10.

*Plik even\_numbers.py:*

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

W powyższym fragmencie kodu działanie funkcji `range()` zaczyna się od wartości 2, a następnie kod dodaje 2 do tej wartości. Proces dodawania 2 jest kontynuowany aż do chwili osiągnięcia lub przekroczenia wartości końcowej (tutaj 11). W wyniku tego procesu zostają wygenerowane następujące dane wyjściowe:

```
[2, 4, 6, 8, 10]
```

Za pomocą funkcji `range()` można utworzyć praktycznie dowolny zbiór liczb. Zastanów się na przykład, jak można utworzyć listę kwadratów pierwszych dziesięciu liczb, to znaczy, jak obliczyć drugą potegę dla liczb całkowitych od 1 do 10. W Pythonie wykładnik jest oznaczany za pomocą dwóch gwiazdek (\*\*). W poniższym fragmencie kodu pokazałem, jak może wyglądać kod umieszczający na liście kwadratów pierwszych dziesięciu liczb.

*Plik squares.py:*

```
squares = []
for value in range(1, 11):
    square = value**2 ❶
    squares.append(square) ❷

print(squares)
```

Pracę rozpoczynamy od utworzenia pustej listy o nazwie `squares`. Następnie nakazujemy Pythonowi, aby przy użyciu funkcji `range()` przeprowadził iterację przez wszystkie wartości od 1 do 10. Wewnątrz bloku pętli bieżąca wartość jest podnoszona do drugiej potęgi i umieszczana w zmiennej `square` (patrz wiersz ❶). Później w wierszu ❷ nowa wartość `square` jest dodawana do listy `squares`. Kiedy pętla zakończy działanie, lista obliczonych kwadratów kolejnych dziesięciu liczb zostaje wyświetlona:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Aby omówiony kod zapisać w jeszcze bardziej zwięzlej postaci, można poinać zmienną tymczasową `square` i nową wartość dodawać bezpośrednio do listy:

---

```
squares = []
for value in range(1, 11):
    squares.append(value**2)

print(squares)
```

---

Kod przedstawiony w tym przykładzie wykonuje takie same operacje jak te w programie przedstawionym w pliku `squares.py`. Każda wartość w pętli jest podnoszona do drugiej potęgi i natychmiast dodawana do listy kwadratów.

Podeczas tworzenia skomplikowanych list możesz użyć dowolnego z obu przedstawionych tutaj rozwiązań. Czasami zastosowanie zmiennej tymczasowej powoduje, że kod staje się łatwiejszy do odczytania, natomiast w innych sytuacjach niepotrzebnie go rozwleka. Przede wszystkim skoncentruj się jednak na tworzeniu czytelnego kodu, którego działanie będzie zgodne z oczekiwaniemi. Następnie podeczas późniejszej analizy kodu rozważ, czy możesz wykorzystać bardziej efektywne podejście do rozwiązania danego problemu.

## Proste dane statystyczne dotyczące listy liczb

Python oferuje kilka funkcji charakterystycznych dla list liczb. Na przykład bardzo łatwo można wskazać najmniejszą i największą liczbę oraz określić sumę liczb z listy:

---

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

---

**UWAGA** Przykłady przedstawione w tym podrozdziale wykorzystują krótkie listy liczb, aby łatwo mieściły się na stronie drukowanej książki. Jednak omawiane funkcje będą działały również dobrze w przypadku list zawierających nawet więcej niż milion elementów.

## Lista składana

Przedstawione powyżej podejście do generowania listy `squares` wymaga użycia trzech lub czterech wierszy kodu. Tak zwana *lista składana* (*list comprehension*) pozwala na wygenerowanie dokładnie tej samej listy, ale za pomocą zaledwie jednego wiersza kodu. Lista składana łączy w pojedynczym wierszu kodu pętlę `for`, utworzenie nowego elementu i jego automatyczne dołączenie do listy. Tego

rodzaju listy nie zawsze są przedstawiane początkującym, ale zdecydowałem się na ich umieszczenie tutaj, ponieważ prawdopodobnie spotkasz się z nimi, gdy tylko zaczniesz analizować kod Pythona tworzony przez innych programistów.

W poniższym fragmencie kodu tworzymy tę samą listę kwadratów jak wcześniej, ale tym razem wykorzystujemy do tego listę składaną.

Plik squares.py:

```
squares = [value**2 for value in range(1, 11)]  
print(squares)
```

W celu użycia tej składni najpierw wybierz jasną nazwę dla listy, taką jak `squares`. Następnie otwórz nawias kwadratowy i zdefiniuj wyrażenie dla wartości, które mają zostać umieszczone na liście. W omawianym przykładzie wyrażeniem jest `value**2`, które powoduje podniesienie bieżącej wartości do drugiej potęgi. Dalej umieść pętlę odpowiedzialną za wygenerowanie liczb, które mają być dostarczane do wyrażenia. Na końcu daj nawias zamykający. Pętla `for` w omawianej liście składanej ma postać `for value in range(1, 11)` i dostarcza wyrażeniu `value**2` wartości od 1 do 10. Zwróć uwagę na brak dwukropka na końcu polecenia `for`.

Wynikiem jest dokładnie taka sama lista kwadratów liczb, jaką widziałeś już wcześniej:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Aby nauczyć się tworzyć własne listy składane, potrzeba nieco praktyki, ale gdy opanujesz już budowanie zwykłych list, prawdopodobnie uznasz, że warto podjąć ten wysiłek. Kiedy będziesz używał trzech lub czterech wierszy kodu do wygenerowania listy i uznasz to za niepotrzebnie powtarzane działanie, rozważ wówczas tworzenie list składanych.

## ZRÓB TO SAM

**4.3. Odliczanie do dwudziestu.** Użyj pętli `for` do wyświetlenia liczb od 1 do 20 włącznie.

**4.4. Milion.** Utwórz listę liczb od jednego do miliona, a następnie wyświetl ją za pomocą pętli `for`. (Jeżeli proces wyświetlania danych wyjściowych trwa zbyt długo, zatrzymaj go przez naciśnięcie klawiszy `Ctrl+C` lub po prostu przez zamknięcie okna danych wyjściowych).

**4.5. Sumowanie do miliona.** Utwórz listę liczb od jednego do miliona, a następnie za pomocą funkcji `min()` i `max()` sprawdź, czy lista faktycznie zaczyna się od wartości jeden i kończy na milionie. Ponadto wykorzystaj funkcję `sum()`, aby zobaczyć, jak szybko Python może dodać milion liczb.

**4.6. Listy nieparzyste.** Za pomocą trzeciego argumentu funkcji `range()` utwórz listę liczb nieparzystych z zakresu od 1 do 20, a następnie wyświetl je za pomocą pętli `for`.

**4.7. Trzy.** Utwórz listę liczb od 3 do 30 podniesionych do trzeciej potęgi, a następnie wyświetl zawartość listy za pomocą pętli `for`.

**4.8. Sześciian.** Liczba podniesiona do trzeciej potęgi jest nazywana *sześciianem*. Na przykład sześciian liczby 2 jest zapisywany w Pythonie jako `2**3`. Utwórz listę pierwszych dziesięciu sześciianów (to znaczy liczb od 1 do 10 podniesionych do trzeciej potęgi), a następnie wyświetl je za pomocą pętli `for`.

**4.9. Sześciian za pomocą listy składanej.** Wykorzystaj listę składaną do wygenerowania listy pierwszych dziesięciu sześciianów.

## Praca z fragmentami listy

W rozdziale 3. dowiedziałeś się, jak uzyskać dostęp do pojedynczych elementów listy. Natomiast w tym rozdziale poznajesz sposoby pracy ze wszystkimi elementami listy. Istnieje również możliwość pracy z określona grupą elementów listy, którą Python określa mianem *wycinka* (*slice*).

### Wycinek listy

W celu utworzenia wycinka należy podać indeks pierwszego i ostatniego elementu, z którym chcesz pracować. Podobnie jak to było w przypadku funkcji `range()`, Python zatrzymuje się na elemencie mającym indeks o jeden mniejszy od podanego indeksu końcowego. Aby utworzyć wycinek zawierający trzy pierwsze elementy listy, trzeba użyć indeksu początkowego 0 i końcowego 3, co spowoduje zwrot elementów o indeksach 0, 1 i 2.

Poniższy przykład dotyczy listy graczy w drużynie.

*Plik players.py:*

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']
print(players[0:3])
```

---

Kod powoduje wyświetlenie wycinka listy, który w tym przykładzie składa się z trzech pierwszych graczy. Wygenerowane dane wyjściowe zachowują strukturę listy:

---

```
['karol', 'martyna', 'michał']
```

---

Istnieje możliwość wygenerowania dowolnego podzbioru listy. Jeśli na przykład chcesz otrzymać drugi, trzeci i czwarty element listy, wówczas podczas definiowania wycinka musisz podać indeks początkowy 1 i końcowy 4:

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']
print(players[1:4])
```

---

Tym razem wycinek zaczyna się od elementu `martyna` i kończy na elemencie `florian`:

---

```
['martyna', 'michał', 'florian']
```

---

Jeżeli pominiesz indeks początkowy, Python automatycznie rozpocznie wycinek od początku listy:

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']
print(players[:4])
```

---

Skoro nie został podany indeks początkowy, tworzenie wycinka Python rozpoczęło od początku listy:

---

```
['karol', 'martyna', 'michał', 'florian']
```

---

Podobną składnię można zastosować, jeśli potrzebny jest wycinek obejmujący ostatni element z listy. Jeżeli na przykład chcesz otrzymać wycinek zawierający elementy od trzeciego do końca, wtedy jako indeks początkowy podaj 2 i pomini indeks końcowy:

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']
print(players[2:])
```

---

W tym przypadku Python zwróci wszystkie elementy od trzeciego do ostatniego na liście:

---

```
['michał', 'florian', 'ela']
```

---

Za pomocą przedstawionej powyżej składni można wyświetlić wszystkie elementy, począwszy od dowolnego miejsca na liście aż do jej końca, niezależnie od wielkości listy. Przypomnij sobie o możliwości użycia ujemnej wartości indeksu pozwalającej na zwrot elementu znajdującego się w określonej odległości od końca listy. Dzięki ujemnej wartości indeksu można utworzyć dowolny wycinek

od końca listy. Jeśli na przykład chcesz otrzymać wycinek zawierający trzy ostatnie elementy listy, jako indeks początkowy podaj -3, jak pokazałem w poniższym fragmencie kodu:

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']
print(players[-3:])
```

---

W tym przykładzie wycinek zawiera imiona trzech ostatnich graczy na liście. Jeżeli wielkość listy ulegnie zmianie, notacja `players[-3:]` nadal będzie zwracała trzy ostatnie elementy listy.

**UWAGA** *Istnieje możliwość umieszczenia jeszcze trzeciej wartości w nawiasie kwadratowym definiującym wycinek. W takim przypadku ta wartość wskazuje Pythonowi liczbę elementów do pominięcia między elementami w podanym przedziale.*

## Iteracja przez wycinek

Wycinek można użyć w pętli `for`, jeśli zachodzi potrzeba przeprowadzenia iteracji przez podzbiór elementów listy. W następnym przykładzie przeprowadzamy iterację przez trzy pierwsze elementy i wyświetlamy imiona graczy w prostym zdaniu:

---

```
players = ['karol', 'martyna', 'michał', 'florian', 'ela']

print("Oto trzech pierwszych graczy naszej drużyny:")
for player in players[:3]: ❶
    print(player.title())
```

---

Pętla `for` zdefiniowana w wierszu ❶ przeprowadza iterację jedynie przez trzy pierwsze elementy listy zamiast przez całą listę graczy:

---

```
Oto trzech pierwszych graczy naszej drużyny:
Karol
Martyna
Michał
```

---

Wycinki okazują się niezwykle użyteczne w wielu różnych sytuacjach. Przykładowo kiedy tworzysz grę, ostateczny wynik uzyskany przez gracza możesz umieścić na liście każdorazowo po zakończeniu rozgrywki. Następnie trzy najlepsze wyniki otrzymujesz przez posortowanie w kolejności malejącej elementów zgromadzonych na tej liście i pobranie wycinka obejmującego tylko trzy pierwsze elementy. Z kolei podczas pracy z danymi wycinki można wykorzystać do przetwarzania danych we fragmentach o żądanej wielkości. Natomiast w przypadku

budowania aplikacji internetowej wycinki mogą zostać wykorzystane do wyświetlania artykułów jako serii stron, z których każda będzie zawierała odpowiednią ilość informacji.

## Kopiowanie listy

Bardzo często pracę chcesz rozpoczęć z istniejącą listą i na jej podstawie utworzyć zupełnie nową. Przeanalizujemy teraz proces kopiowania listy i przyjrzymy się jednej z sytuacji, w której kopowanie listy okazuje się użyteczne.

Aby skopiować listę, można utworzyć wycinek zawierający całą listę początkową, co wymaga pominięcia indeksu początkowego i końcowego (`[:]`). W ten sposób nakazujemy Pythonowi utworzenie wycinka rozpoczynającego się od pierwszego elementu listy i kończącego na ostatnim, czyli powstaje kopia całej listy.

Wyobraź sobie na przykład, że istnieje lista Twoich ulubionych potraw. Chciałbyś utworzyć oddzielną listę przeznaczoną na ulubione potrawy przyjaciela. Ponieważ przyjaciel lubi dokładnie to samo co Ty, jego listę stworzysz przez skopowanie własnej.

Plik `foods.py`:

---

```
my_foods = ['pizza', 'falafel', 'ciasto z marchwi']
friend_foods = my_foods[:] ❶

print("Moje ulubione potrawy to:")
print(my_foods)

print("\nUlubione potrawy mojego przyjaciela to:")
print(friend_foods)
```

---

Zaczynamy od utworzenia listy o nazwie `my_foods`, zawierającej ulubione potrawy. Następnie tworzymy nową listę, `friend_foods`. Kopia listy `my_foods` powstaje przez zdefiniowanie wycinka listy bez podania indeksu początkowego i końcowego ❶. Wspomniana kopia zostaje umieszczona w `friend_foods`. Po wyświetleniu obu list dokładnie widać, że są takie same:

---

```
Moje ulubione potrawy to:
['pizza', 'falafel', 'ciasto z marchwi']

Ulubione potrawy mojego przyjaciela to:
['pizza', 'falafel', 'ciasto z marchwi']
```

---

Aby udowodnić, że w rzeczywistości mamy dwie oddzielne listy, do każdej z nich dodajemy nową potrawę. W ten sposób widać, że każda lista zawiera ulubione potrawy poszczególnych osób:

---

```
my_foods = ['pizza', 'falafel', 'ciasto z marchwi']
friend_foods = my_foods[:] ❶

my_foods.append('cannolo') ❷
friend_foods.append('lody') ❸

print("Moje ulubione potrawy to:")
print(my_foods)

print("\nUlubione potrawy mojego przyjaciela to:")
print(friend_foods)
```

---

Najpierw kopujemy zawartość listy `my_foods` do nowej listy `friend_foods`, podobnie jak w poprzednim przykładzie ❶. Następnie do każdej listy dodajemy nową potrawę: w wierszu ❷ dodajemy '`cannolo`' do `my_foods`, natomiast w wierszu ❸ dodajemy '`lody`' do `friend_foods`. Później wyświetlamy obie listy, aby sprawdzić, czy nowe potrawy znajdują się na odpowiednich listach:

---

```
Moje ulubione potrawy to:
['pizza', 'falafel', 'ciasto z marchwi', 'cannolo']
```

```
Ulubione potrawy mojego przyjaciela to:
['pizza', 'falafel', 'ciasto z marchwi', 'lody']
```

---

Wygenerowane dane wyjściowe pokazują, że element '`cannolo`' pojawił się na liście `my_foods`, natomiast '`lody`' już nie. Dane wyjściowe pokazują również, że lista przyjaciela zawiera nowy element '`lody`', ale już nie '`cannolo`'. W przypadku kiedy po prostu przypiszemy liście `friend_foods` listę `my_foods`, nie otrzymamy dwóch oddzielnych list. W poniższym fragmencie kodu pokazalem sytuację, gdy próbujemy utworzyć kopię listy bez użycia wycinka:

---

```
my_foods = ['pizza', 'falafel', 'ciasto z marchwi']

# Poniższe rozwiązywanie nie działa.
friend_foods = my_foods

my_foods.append('cannolo')
friend_foods.append('lody')

print("Moje ulubione potrawy to:")
print(my_foods)

print("\nUlubione potrawy mojego przyjaciela to:")
print(friend_foods)
```

---

Zamiast otrzymać kopię `my_foods` w `friend_foods`, po prostu wskazujemy, że lista `friend_foods` jest równa liście `my_foods`. Ta składnia tak naprawdę nakazuje Pythonowi połączenie nowej zmiennej `friend_foods` z listą istniejącą już w `my_foods`, więc obie zmienne prowadzą do tej samej listy. Dlatego też kiedy dodamy '`cannolo`' do `my_foods`, nowy element pojawi się również na liście `friend_foods`. Podobnie nowy element '`lody`' pojawi się na obu listach pomimo wrażenia, że został dodany jedynie do `friend_foods`.

Wygenerowane dane wyjściowe pokazują, że obie listy są teraz jednakowe, co nie jest oczekiwany efektem:

---

Moje ulubione potrawy to:

[`'pizza'`, `'falafel'`, `'ciasto z marchwi'`, `'cannolo'`, `'lody'`]

Ulubione potrawy mojego przyjaciela to:

[`'pizza'`, `'falafel'`, `'ciasto z marchwi'`, `'cannolo'`, `'lody'`]

---

**UWAGA** *Na obecnym etapie nie musisz zrozumieć wszystkich szczegółów powyższego przykładu. Ogólnie rzecz biorąc, jeśli próbujesz pracować z kopią listy i spotykasz się z nieoczekiwany zachowaniem programu, sprawdź, czy kopię tej listy stworzyłeś za pomocą wycinka, tak jak pokazałem w pierwszym przykładzie.*

## ZRÓB TO SAM

**4.10. Wycinki.** Pracę rozpoczęj od jednego z programów utworzonych w tym rozdziale, a następnie na końcu dodaj kilka wierszy kodu wykonujących wymienione poniżej zadania:

- Wyświetlenie komunikatu „Pierwsze trzy elementy listy to:”. Następnie za pomocą wycinka wyświetl trzy pierwsze elementy listy.
- Wyświetlenie komunikatu „Trzy elementy w środku listy to:”. Następnie za pomocą wycinka wyświetl trzy elementy znajdujące się w środku listy.
- Wyświetlenie komunikatu „Ostatnie trzy elementy listy to:”. Następnie za pomocą wycinka wyświetl trzy ostatnie elementy listy.

**4.11. Moja pizza, Twoja pizza.** Pracę rozpoczęj od programu utworzonego w ćwiczeniu 4.1 we wcześniejszej części rozdziału. Utwórz kopię listy z pizzas i nadaj jej nazwę `friend_pizzas`. Teraz wykonaj wymienione poniżej zadania:

- Dodaj nową pizzę do listy początkowej.
- Dodaj pizzę (inną niż w poprzednim punkcie) do listy `friend_pizzas`.
- Sprawdź, czy faktycznie masz dwie oddzielne listy. Wyświetl komunikat „Moje ulubione rodzaje pizzy to:”, a następnie użyj pętli `for` do wyświetlenia zawartości pierwszej listy. Teraz wyświetl komunikat „Ulubione rodzaje pizzy mojego przyjaciela to:” i za pomocą pętli `for` wyświetl zawartość drugiej listy. Upewnij się, że każda nowa pizza została umieszczona na odpowiedniej liście.

**4.12. Więcej pętli.** We wszystkich programach *foods.py* przedstawionych w tym rozdziale unikałem użycia pętli *for*, aby zaoszczędzić nieco miejsca. Wybierz dowolną wersję programu *foods.py*, a następnie utwórz dwie pętle *for* w celu wyświetlenie obu list potraw.

# Krotka

List sprawdza się doskonale podczas przechowywania zbioru elementów zmieniających się w trakcie cyklu życiowego programu. Możliwość modyfikowania listy jest szczególnie ważna podczas pracy z listami użytkowników witryny internetowej lub listami postaci w grze. Jednak czasami zachodzi potrzeba utworzenia listy elementów, które nie mogą się zmienić. Tutaj z pomocą przychodzi krotka. W Pythonie wartości, które nie mogą się zmieniać, są określane mianem *niemodyfikowalnych*, a przykładem niemodyfikowalnej listy jest *krotka*.

## Definiowanie krotki

Krotka wygląda niemal tak samo jak lista, ale stosuje nawias okrągły zamiast kwadratowego. Po zdefiniowaniu krotki dostęp do poszczególnych jej elementów uzyskujemy za pomocą indeksu, czyli podobnie jak w przypadku listy.

Przyjmujemy założenie o istnieniu prostokąta, który zawsze powinien mieć określoną wielkość. Aby zagwarantować niezmienność wielkości prostokąta, jego wymiary umieszczamy w krotce, tak jak pokazałem poniżej.

*Plik dimensions.py:*

---

```
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

---

W kodzie zdefiniowaliśmy krotkę o nazwie `dimensions`, używając do tego nawiasu okrągłego, a nie kwadratowego jak podczas tworzenia listy. Następnie rozpoczynamy wyświetlanie poszczególnych elementów krotki za pomocą tej samej składni indeksu, którą stosujemy podczas uzyskiwania dostępu do elementów listy:

---

```
200
50
```

---

Zobaczmy, co się stanie w sytuacji, gdy spróbujemy zmienić jeden z elementów krotki `dimensions`:

---

```
dimensions = (200, 50)
dimensions[0] = 250
```

---

Kod próbuje zmienić wartość pierwszego elementu krotki, a Python zgłasza błąd typu. Ogólnie rzecz biorąc, próbujemy zmienić krotkę, co jest niedozwolone dla obiektu tego typu. Dlatego też Python informuje o braku możliwości przypisania nowej wartości elementowi krotki:

---

```
Traceback (most recent call last):
  File "dimensions.py", line 3, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

---

Taka sytuacja jest dla nas korzystna, ponieważ chcemy, aby Python zgłosił błąd, gdy wiersz kodu spróbuje zmienić wymiary prostokąta.

**UWAGA** *Formalnie rzecz biorąc, krotka jest definiowana po umieszczeniu przecinka w nawiasie. Obecność nawiasu ma poprawić wygląd i czytelność krotki. Jeżeli chcesz zdefiniować krotką jednoelementową, musisz pamiętać o umieszczeniu przecinka po jej elemencie:*

---

```
my_t = (3,)
```

---

*Wprawdzie tworzenie krotki zawierającej tylko jeden element często nie ma sensu, ale może się to zdarzyć podczas automatycznego generowania krotki.*

## Iteracja przez wszystkie wartości krotki

Iterację przez wszystkie wartości krotki można przeprowadzić za pomocą pętli for, czyli podobnie jak w przypadku listy:

---

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

---

Python zwraca wszystkie elementy krotki, podobnie jak to było w przypadku listy:

---

```
200
50
```

---

## Nadpisanie krotki

Wprawdzie nie można zmodyfikować krotki, ale istnieje możliwość przypisania zupełnie nowej wartości zmiennej przechowującej tę krotkę. Dlatego też jeśli chcemy zmienić wymiary prostokąta, możemy od nowa zdefiniować całą krotkę:

---

```
dimensions = (200, 50)
print("Wymiary początkowe:")
for dimension in dimensions:
    print(dimension)

dimensions = (400, 100)
print("\nWymiary po modyfikacji:")
for dimension in dimensions:
    print(dimension)
```

---

Ten blok kodu definiuje pierwotną krotkę i wyświetla jej wymiary początkowe. Na początku umieszczamy zupełnie nową krotkę w zmiennej `dimensions`. Następnie wyświetlamy nowe wymiary prostokąta. Tym razem Python nie zgłasza żadnych błędów, ponieważ nadpisanie zmiennej krotki jest dozwolone:

---

```
Wymiary początkowe:
200
50
```

```
Wymiary po modyfikacji:
400
100
```

---

W porównaniu z listami krotki są prostymi strukturami danych. Wykorzystuj je wtedy, gdy chcesz przechowywać wartości, które nie powinny się zmieniać w trakcie cyklu życiowego programu.

### ZRÓB TO SAM

**4.13. Bufet.** Restauracja w stylu bufetu oferuje jedynie pięć prostych potraw. Wymień więc pięć prostych potraw i umieść je w krotce.

- Za pomocą pętli `for` wyświetl wszystkie potrawy oferowane przez restaurację.
- Spróbuj zmodyfikować jeden z elementów i upewnij się, że Python odrzuci tę zmianę.
- Restauracja zmienia menu i zastępuje dwie dotychczasowe potrawy nowymi. Dodaj blok kodu nadpisujący krotkę, a następnie wykorzystaj pętlę `for` do wyświetlania wszystkich potraw ze zmodyfikowanego menu.

## Styl tworzonego kodu

Teraz, gdy zaczynasz już tworzyć dłuższe programy, warto przez chwilę zastanowić się nad stylem kodu źródłowego. Poświęć nieco czasu i zmodyfikuj kod w taki sposób, aby był jak najbardziej czytelny. Tworzenie łatwego do odczytania

kodu pomaga w zrozumieniu sposobu działania programu nie tylko Tobie, lecz także innym programistom.

Programiści Pythona zgodzili się na zastosowanie wielu konwencji stylu, aby zagwarantować, że tworzony przez nich kod źródłowy będzie miał mniej więcej taki sam styl. Kiedy nauczysz się tworzyć przejrzysty kod Pythona, wówczas będziesz umiał też zrozumieć ogólną strukturę kodu Pythona utworzonego przez innych programistów, o ile zastosowali się oni do tych samych konwencji. Jeżeli marzysz o karierze profesjonalnego programisty Pythona, jak najwcześniej powinieneś zacząć stosować się do wspomnianych konwencji, aby wyrobić w sobie dobre nawyki.

## Konwencje stylu

Kiedy ktoś chce wprowadzić zmianę w języku Python, zaczyna od otworzenia dokumentu nazywanego PEP (ang. *Python Enhancement Proposal*). Jednym z najstarszych dokumentów PEP jest *PEP 8*, który zawiera informacje o sposobach nadawania stylu tworzonemu kodowi Pythona. Dokument *PEP 8* jest dość obszerny, ale jego większa część odnosi się do bardziej złożonych struktur kodu niż tworzone dotąd w tej książce.

Konwencje stylu kodu Pythona zostały opracowane z uwzględnieniem tego, że znacznie częściej odczytujemy kod, niż go tworzymy. Kod tworzysz raz, a następnie zaczynasz go odczytywać po rozpoczęciu fazy debugowania. Kiedy dodajesz następne funkcje do programu, wówczas również więcej czasu poświęcasz na odczytywanie kodu. Gdy udostępniasz kod innym programistom, oni również go czytają.

Mając do wyboru tworzenie kodu łatwiejszego do zapisania lub do odczytywania, programiści Pythona niemal zawsze będą zachęcać Cię do tworzenia kodu, który jest łatwiejszy do odczytu. Przedstawione poniżej wskazówki pomogą Ci w pisaniu kodu, który od samego początku będzie przejrzysty.

## Wcięcia

Specyfikacja *PEP 8* zaleca użycie czterech spacji jako jednego poziomu wcięcia. Zastosowanie czterech spacji poprawia czytelność kodu i jednocześnie pozostawia miejsce na wiele poziomów wcięć możliwych do użycia w każdym wierszu.

W dokumencie procesora tekstu do tworzenia wcięć użytkownicy bardzo często używają tabulatorów zamiast spacji. Wprawdzie takie rozwiązanie sprawdza się doskonale w dokumentach procesora tekstu, ale interpreter Pythona będzie zdezorientowany, gdy zaczniesz łączyć tabulatory i spacje. Każdy edytor tekstu zawiera ustawienie pozwalające na konwersję naciśnięcia klawisza tabulatora na wskazaną liczbę spacji. Zdecydowanie powinieneś używać klawisza tabulatora, ale upewnij się, że konfiguracja edytora powoduje wówczas wstawienie w dokumencie spacji zamiast tabulatora.

Łączenie w jednym pliku tabulatorów i spacji może spowodować powstanie problemów, które będą trudne do zdiagnozowania. Jeżeli sądzisz, że masz do czynienia z połączeniem tabulatorów i spacji, to możesz skorzystać z oferowanej przez większość edytorów funkcji przeprowadzającej konwersję wszystkich tabulatorów w pliku na spacje.

## Długość wiersza

Wielu programistów Pythona zaleca, aby wiersz kodu miał mniej niż 80 znaków. To zalecenie ma swoje korzenie w początkach informatyki, ponieważ większość komputerów mogło wówczas zmieścić jedynie 79 znaków w pojedynczym wierszu okna terminala. Obecnie użytkownicy mogą zmieścić znacznie więcej znaków w wierszu kodu wyświetlany na ekranie, ale istnieją jeszcze inne powody ograniczenia długości wiersza do 79 znaków.

Profesjonalni programiści często mają otwartych wiele plików na tym samym ekranie, więc standardowa długość wiersza pozwala im na jednocześnie wyświetlanie na ekranie dwóch lub trzech plików obok siebie. Specyfikacja PEP zaleca również ograniczenie wszystkich komentarzy do 72 znaków w wierszu, ponieważ część narzędzi, które automatycznie generują dokumentację w ogromnych projektach, dodaje znaki formatowania na początku każdego wiersza komentarza.

Przedstawione w specyfikacji PEP 8 zalecenia dotyczące długości wiersza nie są ustalone na wieki i część zespołów preferuje limit 99 znaków dla wiersza. Podczas nauki programowania nie przejmuj się zbyt bardzo długością wiersza, choć jednocześnie powinieneś mieć świadomość, że gdy więcej osób pracuje nad projektem, wówczas prawie zawsze stosowane są zalecenia PEP 8. Większość edytorów pozwala na ustawienie wizualnego znaku (wykole w postaci pionowej linii na ekranie), który będzie wskazywał zdefiniowany limit długości wiersza.

**UWAGA** *W dodatku B pokazuję, jak skonfigurować edytor tekstu, aby zawsze wstawiał cztery spacje po naciśnięciu klawisza tabulacji oraz wyświetlał pionową linię pomagającą w stosowaniu się do ograniczenia długości wiersza do 79 znaków.*

## Puste wiersze

W celu wizualnego grupowania fragmentów programu używamy pustych wierszy. Wprawdzie powinieneś używać pustych wierszy do organizacji kodu w pliku, ale na pewno nie możesz ich nadużywać. Analizując przykłady przedstawione w książce, powinieneś umieć określić odpowiednią ilość pustych wierszy w kodzie. Przykładowo jeśli masz pięć wierszy kodu tworzących listę, a następnie trzy wiersze wykonujące operacje na tej liście, wówczas rozsądne wydaje się umieszczenie pustego wiersza między wymienionymi fragmentami. Jednak nie powinieneś umieszczać trzech lub czterech pustych wierszy między dwiema sekcjami kodu.

Puste wiersze nie mają wpływu na sposób działania kodu, ale zdecydowanie wpływają na jego czytelność. Interpreter Pythona używa poziomych wcięć do interpretacji znaczenia kodu i zupełnie ignoruje wcięcia pionowe.

## Inne specyfikacje stylu

Specyfikacja PEP 8 zawiera jeszcze inne zalecenia dotyczące stylu, ale większość z nich odnosi się do bardziej skomplikowanych programów niż tworzone przez nas na tym etapie. Gdy będziesz poznawał coraz bardziej złożone struktury Pythona, wtedy będę prezentował związane z nimi zalecenia, które są zdefiniowane w PEP 8.

## ZRÓB TO SAM

**4.14. Specyfikacja PEP 8.** Przejrzyj specyfikację PEP 8, którą znajdziesz na stronie <https://www.python.org/dev/peps/pep-0008/>. W tym momencie nie-wiele z niej skorzystasz, ale i tak warto ją przynajmniej przejrzeć.

**4.15. Przegląd kodu.** Wybierz trzy dowolne programy z tych, które przygotowałeś w tym rozdziale, a następnie zmodyfikuj je, aby stały się zgodne z PEP 8.

- Używaj czterech spacji dla każdego poziomu wcięć. Jeżeli jeszcze tego nie zrobiłeś, skonfiguruj edytor tekstu w taki sposób, aby wstawiał cztery spacje za każdym razem, gdy naciśniesz klawisz tabulacji (więcej informacji na ten temat znajdziesz w dodatku B).
- Używaj mniej niż 80 znaków w każdym wierszu. W edytorze tekstu włącz wyświetlanie pionowej linii wskazującej położenie 80. znaku.
- W programie oszczędnie używaj pustych wierszy.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak można efektywnie pracować z elementami znajdującymi się na liście. Zobaczyłeś, jak przeprowadzić iterację przez listę za pomocą pętli `for`, jak Python używa wcięć do nadania struktury programu oraz jak unikać najczęściej występujących błędów związanych z wcięciami. Nauczyłeś się tworzyć proste listy liczbowe oraz przeprowadzać na nich operacje. Dowiedziałeś się, jak utworzyć wycinek listy w celu pracy z podzbiorem elementów oraz jak prawidłowo skopiować listę za pomocą wycinka. Ponadto poznaleś krotki oferujące pewien poziom bezpieczeństwa dzięki zdefiniowaniu wartości, które nie powinny się zmieniać. Na koniec zajęliśmy się tematem stosowania odpowiedniego stylu w kodzie źródłowym, aby był jak najłatwiejszy do odczytania.

W rozdziale 5. nauczysz się odpowiednio reagować na różne warunki, używając do tego konstrukcji `if`. Zobaczyż, jak można połączyć ze sobą względnie skomplikowane zbiory testów warunkowych, aby reagować odpowiednio do sytuacji lub rodzaju szukanych danych. Ponadto pokażę Ci, jak używać konstrukcji `if` podczas iteracji przez listę w celu wykonania właściwych operacji na wybranych elementach listy.

# 5

## Konstrukcja if



PROGRAMOWANIE CZĘSTO OBEJMUJE PRZEANALIZOWANIE ZESTAWU WARUNKÓW I USTALENIE NA ICH PODSTAWIE, JAKA POWINNA ZOSTAĆ PODJĘTA AKCJA. KONSTRUKCJA `if` W PYTHONIE POZWALA DOKONAĆ ANALIZY BIEŻĄCEGO STANU W PROGRAMIE I ODPOWIĘDNIOW ZAREAGOWAĆ Z UWZGLĘDNIENIEM TEGO STANU.

W tym rozdziale zobaczysz, jak można tworzyć testy warunkowe pozwalające na sprawdzenie praktycznie dowolnego warunku. Dowiesz się, jak tworzyć zarówno proste konstrukcje `if`, jak i te bardziej zaawansowane, które pomagają ustalić, kiedy dokładnie wystąpił interesujący nas warunek. Następnie tę koncepcję zastosujemy na listach, więc będziesz mógł utworzyć pętlę `for` obsługującą wiele różnych elementów listy w jeden sposób, a w zupełnie inny sposób obsługującą wybrane elementy o określonych wartościach.

### Prosty przykład

Poniższy krótki fragmentu kodu pokazuje, jak konstrukcja `if` pozwala prawidłowo zareagować w pewnej specjalnej sytuacji. Przyjmujemy założenie, że mamy listę samochodów i chcemy wyświetlić markę każdego z nich. Nazwy samochodów to prawidłowe nazwy marek, więc praktycznie zawsze powinny być wyświetlane z użyciem wielkiej litery na początku. Istnieją jednak pewne wyjątki i na przykład wartość '`bmw`' powinna być cała wyświetlona wielkimi literami. Kod w poniższym programie przeprowadza iterację przez listę marek samochodów i szuka wartości '`bmw`'. Po jej odszukaniu cała nazwa zostaje zapisana wielkimi literami, a nie tylko jej pierwsza litera.

Plik cars.py:

---

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
    if car == 'bmw': ❶
        print(car.upper())
    else:
        print(car.title())
```

---

Pętla w wierszu ❶ sprawdza, czy wartością bieżącą zmiennej `car` jest '`bmw`'. Jeżeli tak, cała wartość będzie wyświetlona wielkimi literami. Natomiast jeśli wartość sprawdzanej zmiennej jest inna niż '`bmw`', tylko pierwsza litera zostanie wyświetlona jako wielka:

---

```
Audi
BMW
Subaru
Toyota
```

---

W powyższym programie wykorzystałem wiele różnych koncepcji, które poznasz w tym rozdziale. Rozpoczynamy od analizy rodzajów testów, jakie można przeprowadzić podczas sprawdzania warunków w programie.

## Test warunkowy

Sercem każdej konstrukcji `if` jest wyrażenie, które może przyjąć wartość `True` lub `False` i jest określane mianem *testu warunkowego*. Python wykorzystuje wartości `True` i `False` do ustalenia, czy kod w bloku `if` powinien zostać wykonany. Jeżeli test warunkowy przyjmuje wartość `True`, Python wykonuje blok kodu umieszczony po poleceniu `if`. Natomiast jest wynikiem testu jest `False`, Python po prostu zignoruje blok kodu znajdujący się po poleceniu `if`.

### Sprawdzenie równości

Większość testów warunkowych porównuje bieżącą wartość zmiennej z inną, szczegółowo nas interesującą wartością. Najprostszy test warunkowy sprawdza, czy wartość zmiennej jest równa interesującej nas wartości:

---

```
>>> car = 'bmw'
>>> car == 'bmw'
True
```

---

W pierwszym wierszu mamy przypisanie zmiennej car wartości 'bmw', co odbywa się za pomocą pojedynczego znaku równości, jak mogłeś wcześniej zobaczyć już nie jeden raz. Natomiast w drugim wierszu sprawdzamy, czy zmienna car ma wartość 'bmw' — tym razem używamy dwóch znaków równości (==). Ten operator równości zwraca wartość True, jeśli po obu stronach operatora znajdują się te same wartości. W przeciwnym razie operator zwraca wartość False. Ponieważ w omawianym przykładzie wartości są takie same, Python zwraca True.

Jeżeli wartość zmiennej car będzie inna niż 'bmw', wówczas wynikiem działania operatora jest False:

---

```
>>> car = 'audi'  
>>> car == 'bmw'  
False
```

---

Pojedyńczy znak równości to tak naprawdę polecenie. Kod w pierwszym wierszu można odczytać jako: „Zmiennej car przypisz wartość równą 'audi'”. Z kolei dwa znaki równości, takie jak te użyte w drugim wierszu, oznaczają pytanie: „Czy wartość zmiennej car jest równa 'bmw'?”. W większości języków programowania znaki równości są używane w taki właśnie sposób.

## Ignorowanie wielkości liter podczas sprawdzania równości

Podeczas sprawdzania równości w Pythonie wielkość liter ma znaczenie. Dlatego też dwie wartości o odmiennej wielkości znaków są uznawane za nierówne.

---

```
>>> car = 'Audi'  
>>> car == 'audi'  
False
```

---

Jeżeli wielkość znaków ma znaczenie, wówczas takie zachowanie domyślne jest oczywiście korzystne. Natomiast gdy wielkość znaków nie ma znaczenia i po prostu chcesz przetestować wartość zmiennej, wówczas możesz ją skonwertować do postaci zapisanej małymi literami i dopiero wtedy porównać:

---

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True
```

---

Powyższy kod zwraca wartość True niezależnie od tego, jak wartość 'Audi' zostanie zapisana, ponieważ teraz wielkość liter nie ma już żadnego znaczenia. Metoda lower() nie zmienia wartości pierwotnie przechowywanej w zmiennej car, więc tego rodzaju porównanie można przeprowadzać bez obaw o zmodyfikowanie pierwotnej zmiennej:

---

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True  
>>> car  
'Audi'
```

---

Zaczynamy od umieszczenia wartości 'Audi' w zmiennej car. Następnie konwertujemy tę wartość do postaci zapisanej małymi literami i porównujemy z ciągiem tekstowym 'audi'. Ponieważ oba ciągi tekstowe są takie same, Python zwraca wartość True. Ten przykład pokazuje, że przeprowadzony test nie miał żadnego wpływu na wartość przechowywaną w zmiennej car.

Witryny internetowe w podobny sposób wymuszają stosowanie pewnych reguł dotyczących danych wprowadzanych przez użytkowników. Przykładowo witryna może używać podobnego testu warunkowego do sprawdzenia, czy podana została naprawdę unikatowa nazwa użytkownika, a nie jedynie zapisany inną wielkością liter wariant już istniejącej nazwy użytkownika. Kiedy dana osoba wprowadza nazwę użytkownika, nazwa ta zostaje skonwertowana do postaci zapisanej małymi literami, a następnie porównana ze wszystkimi istniejącymi nazwami. W trakcie tego rodzaju operacji nazwa użytkownika 'Janek' zostanie odrzucona, jeśli istnieje już nazwa użytkownika 'janek'.

## Sprawdzenie nierówności

Kiedy chcesz ustalić, czy dwie wartości są nierówne, możesz użyć *operatora nierówności* (!=). Wykorzystamy teraz inne polecenie if do pokazania sposobu użycia operatora nierówności. W zmiennej umieścimy oczekiwane dodatki do pizzy, a następnie wyświetlimy komunikat, jeśli klient nie zamówił pizzy z anchois.

Plik toppings.py:

---

```
requested_topping = 'pieczarki'  
  
if requested_topping != 'anchois':  
    print("Proszę o anchois!")
```

---

W kodzie mamy porównanie wartości zmiennej requested\_topping z wartością 'anchois'. Jeżeli te dwie wartości będą różne, wówczas Python zwróci True i wykona kod znajdujący się tuż po poleceniu if. Natomiast w przypadku kiedy obie wartości będą takie same, Python zwróci False i nie wykona bloku kodu po poleceniu if.

Ponieważ wartością zmiennej requested\_topping nie jest 'anchois', wywołana zostaje funkcja print():

---

```
Proszę o anchois!
```

---

Większość wyrażeń warunkowych będzie sprawdzało równość, choć czasami się przekonasz, że efektywniejsze może się okazać sprawdzenie pod kątem nierówności.

## Porównania liczbowe

Sprawdzanie wartości liczbowych jest całkiem proste. Przykładowo przedstawiony poniżej fragment kodu sprawdza, czy osoba ma 18 lat:

```
>>> age = 18
>>> age == 18
True
```

Istnieje również możliwość sprawdzenia, czy dwie liczby są nierówne. Dlatego też poniższy fragment kodu powoduje wyświetlenie komunikatu w przypadku udzielenia nieprawidłowej odpowiedzi.

Plik `magic_number.py`:

```
answer = 17
if answer != 42:
    print("To nie jest prawidłowa odpowiedź. Proszę spróbuj ponownie!")
```

Warunek sprawdzany w kodzie jest prawdziwy, ponieważ wartość zmiennej `answer` (tutaj wynosi 17) nie jest równa 42. Skoro wynikiem testu jest wartość `True`, zostaje wykonany blok polecenia `if`:

```
To nie jest prawidłowa odpowiedź. Proszę spróbuj ponownie!
```

W poleceniach warunkowych można również umieszczać operacje matematyczne, na przykład mniejszy niż, mniejszy niż lub równy, większy niż oraz większy niż lub równy:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Jeżeli tego rodzaju operacja matematyczna zostanie użyta jako fragment polecenia `if`, może pomóc w wykryciu konkretnych interesujących Cię warunków.

## Sprawdzanie wielu warunków

Być może zajdzie konieczność jednoczesnego sprawdzenia wielu warunków. Przykładowo czasami dwa warunki muszą zwrócić wartość True, abyś mógł podjąć akcję. Innym razem możesz być usatysfakcjonowany tylko jednym warunkiem zwracającym wartość True. W tego rodzaju sytuacjach mogą pomóc słowa kluczowe and i or.

### Użycie słowa kluczowego and do sprawdzania wielu warunków

Aby sprawdzić, czy dwa warunki jednocześnie zwracają wartość True, należy użyć słowa kluczowego and i za jego pomocą połączyć dwa testy. Jeżeli oba testy zostaną zaliczone, całe wyrażenie zwróci wartość True. Natomiast jeśli chociażby jeden z nich nie zostanie zaliczony, całe wyrażenie zwróci wartość False.

Przykładowo w poniższym fragmencie kodu sprawdzamy, czy obie osoby mają co najmniej 21 lat:

---

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21 ❶
False
>>> age_1 = 22 ❷
>>> age_0 >= 21 and age_1 >= 21
True
```

---

Zaczynamy od zdefiniowania dwóch zmiennych `age_0` i `age_1`. Później w wierszu ❶ sprawdzamy, czy obie wartości są większe niż 21. Test po lewej stronie wyrażenia zostaje zaliczony, natomiast po prawej stronie nie. Dlatego też wartością całego wyrażenia jest `False`. W wierszu ❷ zmieniamy wartość `age_1` na 22. W tym momencie wartość zmiennej `age_1` jest większa niż 21, więc oba testy zostają zaliczone, a wartością całego wyrażenia jest `True`.

Aby poprawić czytelność kodu, poszczególne testy można ująć w nawias, choć to nie jest wymagane. Jeżeli zdecydujesz się na użycie nawiasów, polecenie będzie miało następującą postać:

---

```
(age_0 >= 21) and (age_1 >= 21)
```

---

### Użycie słowa kluczowego or do sprawdzania wielu warunków

Słowo kluczowe or również pozwala sprawdzić wiele warunków, ale tym razem tylko jeden z nich musi być spełniony. Wartością całego wyrażenia będzie `False` tylko wtedy, gdy oba pojedyncze testy zakończą się niepowodzeniem.

Ponownie przechodzimy do przykładu z wiekiem osób. Tym razem tylko jedna z nich musi mieć co najmniej 21 lat:

---

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21 ❶
True
>>> age_0 = 18 ❷
>>> age_0 >= 21 or age_1 >= 21
False
```

---

Rozpoczynamy od zdefiniowania dwóch zmiennych. Ponieważ w wierszu ❶ zaliczany jest test dla zmiennej `age_0`, całe wyrażenie zwraca wartość `True`. Następnie wiek zapisany w zmiennej `age_0` obniżamy do zaledwie 18. Przeprowadzamy testy (patrz wiersz ❷) — oba kończą się niepowodzeniem i wartością całego wyrażenia jest `False`.

## Sprawdzanie, czy wartość znajduje się na liście

Czasami ważne jest sprawdzenie jeszcze przed podjęciem akcji, czy lista zawiera określoną wartość. Przykładowo zanim w witrynie internetowej zostanie zakończony proces rejestracji użytkownika, trzeba sprawdzić, czy wybrana przez tego użytkownika nazwa już znajduje się na liście aktualnych nazw użytkowników. Z kolei w projekcie kartograficznym należy sprawdzić, czy przekazana lokalizacja znajduje się już na liście znanych lokalizacji.

W celu ustalenia, czy określona wartość znajduje się na liście, należy użyć słowa kluczowego `in`. Przeanalizujmy pewien kod opracowany dla pizzerii. Tworzymy listę wybranych przez klienta dodatków do pizzy, a następnie sprawdzamy, czy określone dodatki znajdują się na tej liście:

---

```
>>> requested_toppings = ['pieczarki', 'cebula', 'ananas']
>>> 'pieczarki' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

---

Słowo kluczowe `in` nakazuje Pythonowi sprawdzenie, czy dodatki — odpowiednio pieczarki i pepperoni — znajdują się na liście `requested_toppings`. Tego rodzaju technika daje potężne możliwości, ponieważ pozwala tworzyć listy istotnych wartości, a następnie łatwo sprawdzać, czy interesująca nas wartość została dopasowana do jakiejkolwiek wartości z listy.

## Sprawdzanie, czy wartość nie znajduje się na liście

Czasami trzeba koniecznie sprawdzić, czy wartość nie pojawia się na liście. W takiej sytuacji z pomocą przychodzi słowo kluczowe `not`. Rozważ na przykład listę użytkowników, którzy mają zakaz publikowania komentarzy na forum. W poniższym fragmencie kodu, zanim umożliwimy użytkownikowi umieszczenie komentarza, sprawdzamy, czy przypadkiem nie ma on zakazu publikowania na forum.

Plik banned\_users.py:

---

```
banned_users = ['andrzej', 'karolina', 'dawid']
user = 'maria'

if user not in banned_users:
    print(f"{user.title()}, możesz opublikować odpowiedź, jeśli chcesz.")
```

---

Działanie przedstawionego tutaj polecenia if powinno być jasne. Jeżeli wartość zmiennej user nie znajduje się na liście banned\_users, Python zwraca wartość True i wykonuje wcięty wiersz kodu.

Ponieważ użytkownik 'maria' nie znajduje się na liście banned\_users, zostaje wyświetlony komunikat zachęcający do opublikowania odpowiedzi:

---

```
Maria, możesz opublikować odpowiedź, jeśli chcesz.
```

---

## Wyrażenie boolowskie

W trakcie nauki programowania na pewno spotkasz się z terminem *wyrażenie boolowskie*. Tego rodzaju wyrażenie to po prostu jeszcze inna nazwa testu warunkowego. Wartością boolowską jest True lub False, podobnie jak w przypadku wyniku testu warunkowego.

Wartości boolowskie są bardzo często używane do śledzenia określonych warunków, na przykład sprawdzania, czy gra nadal trwa lub czy użytkownik może edytować pewną treść w witrynie internetowej:

---

```
game_active = True
can_edit = False
```

---

Wartości boolowskie umożliwiają efektywne monitorowanie stanu programu lub określonego warunku, który jest niezwykle ważny w programie.

### ZRÓB TO SAM

**5.1. Testy warunkowe.** Utwórz serię testów warunkowych. Wyświetl polecenia opisujące poszczególne testy oraz przewidywany wynik. Przygotowany przez Ciebie kod powinien być podobny do poniższego:

```
car = 'subaru'
print("Czy car == 'subaru'? Przewiduję wartość True.")
print(car == 'subaru')

print("\nCzy car == 'audi'? Przewiduję wartość False.")
print(car == 'audi')
```

---

- Przyjrzyj się dokładnie uzyskanym wynikom i upewnij się, że zrozumiałeś, dlaczego wartością danego testu jest True lub False.
- Utwórz przynajmniej 10 testów. Niech co najmniej pięć z nich przyjmuje wartość True, a kolejne pięć wartość False.

**5.2. Więcej testów warunkowych.** Nie musisz ograniczać się tylko do 10 testów. Jeżeli chcesz wypróbować inne porównania, utwórz następne testy i umieść je w pliku *conditional\_tests.py*. Powinieneś otrzymać co najmniej po jednym wyniku True i False dla wymienionych poniżej testów:

- Sprawdzenie równości i nierówności ciągów tekstowych.
- Test z użyciem metody `lower()`.
- Testy liczbowe obejmujące sprawdzenie równości i nierówności, operacje większy niż i mniejszy niż, a także większy niż lub równy i mniejszy niż lub równy.
- Testy wykorzystujące słowa kluczowe `and` i `or`.
- Sprawdzanie, czy element znajduje się na liście.
- Sprawdzanie, czy element nie znajduje się na liście.

## Polecenie if

Kiedy zrozumiesz sposób działania testów warunkowych, możesz przystąpić do tworzenia własnych poleceń `if`. Istnieje wiele różnych rodzajów poleceń `if`, a wybór konkretnego zależy od liczby warunków koniecznych do przetestowania. Wprawdzie kilka przykładów poleceń `if` mogłeś już zobaczyć przy okazji omawiania testów warunkowych, ale teraz zajmiemy się tym nieco dokładniej.

### Proste polecenia if

Poniżej przedstawiłem najprostszy rodzaj polecenia `if`, które zawiera tylko jeden test i jedną akcję:

---

```
if test_warunkowy:  
    dowolna akcja
```

---

Dowolny test warunkowy umieszczaemy w wierszu pierwszym, natomiast dowolną akcję we wciętym bloku kodu znajdującym się tuż za testem. Jeżeli wynikiem testu warunkowego będzie `True`, Python wykona fragmentu kodu znajdujący się tuż po poleceniu `if`. Natomiast wartość `False` testu powoduje, że Python zignoruje blok kodu po poleceniu `if`.

Przyjmujemy założenie, że mamy zmienną przechowującą wiek osoby i chcemy ustalić, czy ta osoba ma wystarczającą liczbę lat, aby wziąć udział w głosowaniu. Poniższy fragment kodu sprawdza, czy dana osoba może uczestniczyć w głosowaniu.

Plik voting.py:

---

```
age = 19
if age >= 18:
    print("Możesz wziąć udział w głosowaniu!")
```

---

Python sprawdza, czy wartość zmiennej `age` jest większa lub równa 18. Jeżeli tak, zostanie wykonane polecenie `print()`, które znajduje się we wciętym wierszu:

---

```
Możesz wziąć udział w głosowaniu!
```

---

W poleceniach `if` wcięcia odgrywają taką samą rolę jak w przypadku pętli `for`. Wszystkie wcięte wiersze kodu po poleceniu `if` zostaną wykonane, jeżeli test będzie zaliczony. Natomiast jeżeli test nie zostanie zaliczony, Python zignoruje cały blok wciętych wierszy.

W bloku znajdująącym się po poleceniu `if` można umieścić dowolną liczbę wierszy kodu. Do omawianego przykładu dodajemy więc następny wiersz danych wyjściowych, który zawiera pytanie o rejestrację w celu wzięcia udziału w głosowaniu:

---

```
age = 19
if age >= 18:
    print("Możesz wziąć udział w głosowaniu!")
    print("Czy zarejestrowałeś się już, aby móc głosować?")
```

---

Test warunkowy zostaje zaliczony, a skoro oba wiersze `print()` są wcięte, na ekranie zostaną wyświetcone dwa komunikaty:

---

```
Możesz wziąć udział w głosowaniu!
Czy zarejestrowałeś się już, aby móc głosować?
```

---

Jeżeli wartość zmiennej `age` będzie mniejsza niż 18, ten program nie wygeneruje żadnych danych wyjściowych.

## Polecenia if-else

Często chcemy podjąć jedną akcję, jeśli test warunkowy zostanie zaliczony, i zupełnie inną, jeśli nie zostanie zaliczony. Oferowana przez Pythona składnia `if-else` daje nam taką możliwość. Konstrukcja `if-else` jest podobna do prostego polecenia `if`, choć polecenie `else` pozwala na zdefiniowanie akcji (lub zestawu akcji) wykonywanej po niezaliczeniu testu warunkowego.

W poniższym fragmencie kodu osobie dorosłej wyświetlamy przedstawione już wcześniej komunikaty oraz dodajemy nowe dla osób, które jeszcze nie mogą wziąć udziału w głosowaniu:

---

```
age = 17
if age >= 18: ❶
    print("Możesz wziąć udział w głosowaniu!")
    print("Czy zarejestrowałeś się już, aby móc głosować?")
else: ❷
    print("Przykro nam, ale jesteś zbyt młody, aby głosować.")
    print("Możesz się zarejestrować po ukończeniu 18 lat!")
```

---

Jeżeli test warunkowy zdefiniowany w wierszu ❶ zostanie zaliczony, to wykonany zostanie pierwszy wcięty blok kodu. Natomiast jeśli wartością testu warunkowego będzie `False`, wykonane zostaną polecenia z bloku `else`, zdefiniowanego począwszy od wiersza ❷. Ponieważ tym razem zmienna `age` ma wartość mniejszą niż 18, test warunkowy kończy się niepowodzeniem i zostaje wykonany blok kodu `else`:

---

```
Przykro nam, ale jesteś zbyt młody, aby głosować.
Możesz się zarejestrować po ukończeniu 18 lat!
```

---

Powyższy kod działa, ponieważ mamy tylko dwie możliwe sytuacje: wiek danej osoby pozwala jej wziąć udział w głosowaniu lub nie pozwala. Struktura `if-else` sprawdza się doskonale w sytuacji, gdy chcesz, aby Python zawsze podjął jedną z dwóch możliwych akcji. W przypadku prostej konstrukcji `if-else`, takiej jak przedstawiona powyżej, zawsze zostanie wykonana jedna z dwóch akcji.

## Konstrukcja `if-elif-else`

Często trzeba sprawdzić więcej niż tylko dwie możliwe sytuacje. Do tego celu można wykorzystać oferowaną przez Pythona składnię `if-elif-else`. W wymienionej konstrukcji Python wykonuje tylko jeden blok kodu. Poszczególne testy warunkowe są wykonywane tak długo, dopóki jeden z nich nie zostanie zaliczony. Kiedy zostanie zaliczony jeden z testów, wykonany zostanie znajdujący się po nim bloku kodu, a pozostałe testy Python pominie.

W wielu rzeczywistych sytuacjach mamy więcej niż dwa możliwe warunki. Weźmy na przykład park rozrywki, w którym ceny biletów są różne dla poszczególnych grup wiekowych zwiedzających:

- Bilet wstępu jest bezpłatny dla dzieci poniżej 4 lat.
- Bilet wstępu kosztuje 25 zł dla dzieci w wieku od 4 do 18 lat.
- Bilet wstępu dla osób w wieku powyżej 18 lat kosztuje 40 zł.

W jaki sposób można użyć polecenia `if`, aby ustalić cenę biletu wstępu dla osoby w danym wieku? Przedstawiony poniżej fragment kodu porównuje wiek osoby z poszczególnymi grupami wiekowymi, a następnie wyświetla odpowiednią cenę biletu.

*Plik amusement\_park.py:*

---

```
age = 12
if age < 4: ❶
    print("Cena biletu wstępu wynosi 0 zł.")
elif age < 18: ❷
    print("Cena biletu wstępu wynosi 25 zł.")
else: ❸
    print("Cena biletu wstępu wynosi 40 zł.)
```

---

Polecenie `if` w wierszu ❶ sprawdza, czy osoba ma poniżej 4 lat. Jeżeli ten test zostanie zaliczony, Python wyświetli odpowiedni komunikat i pominie przeprowadzenie pozostałych testów w konstrukcji `if`. Zdefiniowane w wierszu ❷ polecenie `elif` to tak naprawdę inny test `if`, który będzie wykonany tylko wtedy, gdy poprzedni zakończy się niepowodzeniem. Na tym etapie wiemy, że dana osoba będzie miała co najmniej 4 lata, ponieważ pierwszy test zakończył się niepowodzeniem. Jeżeli osoba ma mniej niż 18 lat, Python wyświetli odpowiedni komunikat i pominie wykonanie bloku `else`. Gdy oba testy (`if` i `elif`) zakończą się niepowodzeniem, wówczas Python wykona kod umieszczony w bloku `else` (patrz wiersz ❸).

W omawianym przykładzie test zdefiniowany w wierszu ❶ kończy się niepowodzeniem (wartość `False`), więc ten blok kodu nie zostanie wykonany. Jednak drugi test przyjmuje wartość `True` (liczba 12 jest mniejsza od 18), więc zostaje wykonany następujący po nim blok kodu. Wygenerowane dane wyjściowe to pojedynczy komunikat podający cenę biletu wstępu dla osoby w danym wieku:

---

Cena biletu wstępu wynosi 25 zł.

---

W przypadku osób w wieku powyżej 17 lat dwa pierwsze testy nie zostaną zaliczone. Zostanie zatem wykonany blok kodu `else`, co oznacza, że bilet wstępu dla takiej osoby kosztuje 40 zł.

Zamiast wyświetlać cenę biletu wstępu wewnętrz konstrukcji `if-elif-else`, znacznie lepszym rozwiązaniem będzie ustalenie w wymienionej konstrukcji jedynie ceny i umieszczenie w kodzie programu pojedynczego wywołania `print()`, którego treść zostanie wyświetlona już po wykonaniu konstrukcji `if-elif-else`:

---

```
age = 12
if age < 4:
    price = 0
```

---

```
elif age < 18:  
    price = 25  
else:  
    price = 40  
  
print(f"Cena biletu wstępu wynosi {price} zł.")
```

---

We wciętych wierszach zostaje zdefiniowana wartość zmiennej `price` na podstawie wieku danej osoby, podobnie jak w poprzednim przykładzie. Kiedy w konstrukcji `if-elif-else` zostanie ustalona cena biletu wstępu, polecenie `print()`, znajdujące się poza wymienioną konstrukcją, użycie wartości zmiennej `price` do wyświetlenia komunikatu podającego tę cenę.

Ten kod powoduje wygenerowanie takich samych danych wyjściowych jak we wcześniejszych przykładach, ale przeznaczenie konstrukcji `if-elif-else` zostało zawężone. Zamiast ustalać cenę i wyświetlać odpowiedni komunikat, po prostu określamy cenę biletu wstępu. Oprócz tego, że taki przeredagowany kod jest bardziej efektywny, jest również łatwiejszy do modyfikowania niż jego pierwotna wersja. Jeżeli postanowisz zmienić treść komunikatu wyświetlanego użytkownikowi, modyfikację trzeba będzie wprowadzić tylko w jednym wywołaniu `print()` zamiast w trzech oddzielnych.

## Użycie wielu bloków elif

W kodzie można użyć dowolną liczbę bloków `elif`. Przykładowo jeśli park rozrywki postanowi wprowadzić zniżki dla seniorów, będzie można dodać jeszcze jeden test warunkowy w celu ustalenia, czy dana osoba kwalifikuje się do otrzymania nowej zniżki. Przyjmujemy założenie, że po osiągnięciu 65 lat zwiedzający płaci jedynie połowę normalnej ceny biletu:

---

```
age = 12  
  
if age < 4:  
    price = 0  
elif age < 18:  
    price = 25  
elif age < 65:  
    price = 40  
else:  
    price = 20  
  
print(f"Cena biletu wstępu wynosi {price} zł.")
```

---

Większa część kodu pozostała bez zmian. Natomiast drugi blok `elif` służy do sprawdzenia, czy osoba ma poniżej 65 lat. Jeżeli tak, jest zobowiązana kupić bilet w normalnej cenie 40 zł. Zwróć uwagę na konieczność zmiany ceny biletu przy pisywanej w bloku `else`. Teraz podajemy tutaj wartość 20, ponieważ ten blok kodu zostanie wykonany jedynie w przypadku osób, które mają co najmniej 65 lat.

## Pominięcie bloku else

Python nie wymaga umieszczenia bloku `else` na końcu konstrukcji `if-elif`. Czasami istnienie bloku `else` jest użyteczne, z kolei w innych sytuacjach znacznie lepszym rozwiązaniem będzie zastosowanie dodatkowego polecenia `elif` przechytującego szczególnie interesujący nas warunek:

---

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"Cena biletu wstępu wynosi {price} zł.")
```

---

Dodatkowy blok `elif` zdefiniowany ustala cenę biletu wstępu na 20 zł dla osoby w wieku co najmniej 65 lat. Takie rozwiązanie jest znacznie czytelniejsze niż użycie ogólnego bloku `else`. Kiedy wprowadzimy tę zmianę, każdy blok kodu zostanie wykonany, jeśli spełniony zostanie określony warunek.

Blok `else` jest wykorzystywany w sytuacji, gdy nie został spełniony żaden z warunków wcześniej zdefiniowanych za pomocą polecen `if` lub `elif`. Brak spełnienia warunku może wynikać z podania nieprawidłowych danych lub danych o złośliwym działaniu. Jeżeli istnieje konkretny, ostateczny warunek do sprawdzenia, rozważ użycie bloku `elif` i całkowite pominięcie bloku `else`. W ten sposób zyskasz absolutną pewność, że kod będzie wykonywany jedynie po spełnieniu oczekiwanych warunków.

## Sprawdzanie wielu warunków

Konstrukcja `if-elif-else` oferuje olbrzymi wachlarz możliwości, ale jest odpowiednia do użycia tylko wtedy, kiedy spełniony ma być jeden warunek. Gdy tylko Python znajdzie zaliczony test, po prostu pomija pozostałe. Tego rodzaju zachowanie okazuje się korzystne, ponieważ jest efektywne i pozwala na sprawdzanie konstrukcji pod kątem jednego konkretnego warunku.

Jednak czasami ważne jest sprawdzenie wszystkich interesujących nas warunków. W takim przypadku powinieneś użyć serii prostych polecień `if` bez żadnych bloków `elif` lub `else`. Taka technika ma sens, gdy więcej niż tylko jeden warunek może przyjąć wartość `True` i chcesz dla wszystkich takich warunków wykonać jakąś operację.

Powróćmy do przykładu pizzerii. Jeżeli klient zażyczy sobie pizzy z dwoma dodatkami, trzeba będzie się upewnić, że oba zostały uwzględnione w zamówieniu.

## Plik toppings.py:

---

```
requested_toppings = ['pieczarki', 'podwójny ser']

if 'pieczarki' in requested_toppings:
    print("Dodaję pieczarki.")
if 'pepperoni' in requested_toppings: ❶
    print("Dodaję pepperoni.")
if 'podwójny ser' in requested_toppings:
    print("Dodaję podwójny ser.")

print("\nTwoja pizza jest już gotowa!")
```

---

Rozpoczynamy od zdefiniowania listy żądanego dodatków. Pierwsze polecenie if sprawdza, czy klient zażyczył sobie, aby do pizzy zostały dodane pieczarki. Jeżeli tak, następuje wygenerowanie komunikatu potwierdzającego dodanie pieczarek. Sprawdzenie składnika pepperoni (patrz wiersz ❶) to kolejne proste polecenie if, a nie elif lub else, obecne tutaj po to, aby test ten został wykonany niezależnie od wyniku poprzedniego testu. Ostatnie polecenie if sprawdza, czy klient zażyczył sobie podwójnego sera. Także to sprawdzenie jest przeprowadzane niezależnie od wyniku poprzednich testów. Tak przygotowane trzy niezależne testy będą wykonywane w trakcie każdego uruchomienia programu.

Ponieważ wszystkie warunki zostaną sprawdzone, do pizzy zostaną dodane zarówno pieczarki, jak i podwójny ser:

---

```
Dodaję pieczarki.  
Dodaję podwójny ser.  
  
Twoja pizza jest już gotowa!
```

---

Ten kod nie będzie działał prawidłowo w przypadku użycia konstrukcji if-elif-else, ponieważ wówczas wykonywanie konstrukcji zostanie zakończone po zaliczeniu pierwszego testu. Oto jakby przedstawiał się omawiany program po zastosowaniu konstrukcji if-elif-else:

---

```
requested_toppings = ['pieczarki', 'podwójny ser']

if 'pieczarki' in requested_toppings:
    print("Dodaję pieczarki.")
elif 'pepperoni' in requested_toppings:
    print("Dodaję pepperoni.")
elif 'podwójny ser' in requested_toppings:
    print("Dodaję podwójny ser.")

print("\nTwoja pizza jest już gotowa!")
```

---

Sprawdzenie dodatku pieczarki to pierwszy zaliczony test, więc jedynie ten dodatek został dodany do pizzy. Z kolei sprawdzenie pod kątem dodatków 'podwójny ser' i 'pepperoni' nigdy nie zostanie przeprowadzone, ponieważ po zaliczeniu pierwszego testu w konstrukcji `if-elif-else` Python nigdy już nie wykona pozostałych testów. Pierwszy z podanych przez klienta dodatków będzie dodany do pizzy, natomiast pozostałe zostaną pominięte:

---

Dodaję pieczarki.

Twoja pizza jest już gotowa!

---

Podsumowując: jeżeli masz tylko jeden blok kodu do wykonania, zdecyduj się na konstrukcję `if-elif-else`. Natomiast jeśli konieczne jest wykonanie więcej niż tylko jednego bloku kodu, wtedy użyj serii niezależnych poleceń `if`.

## ZRÓB TO SAM

**5.3. Kolory obcych, część 1.** Wyobraź sobie zestrzelenie obcego w grze. Utwórz zmienną o nazwie `alien_color` i przypisz jej wartość 'zielony', 'żółty' lub 'czerwony'.

- Przygotuj polecenie `if` sprawdzające, czy kolorem obcego jest zielony. Jeżeli tak, wyświetl komunikat informujący gracza, że zarobił pięć punktów.
- Przygotuj wersję programu zaliczającą powyższy test oraz drugą niezaliczającą. (Wersja kończąca się niepowodzeniem nie powinna generować żadnych danych wyjściowych).

**5.4. Kolory obcych, część 2.** Podobnie jak w ćwiczeniu 5.3 wybierz kolor dla obcego, a następnie utwórz konstrukcję `if-else`.

- Jeżeli kolor obcego to zielony, wyświetl komunikat informujący gracza, że zarobił pięć punktów za zestrzelenie tego obcego.
- Jeżeli kolor obcego jest inny niż zielony, wyświetl komunikat informujący gracza, że zarobił dziesięć punktów za zestrzelenie tego obcego.
- Przygotuj wersję programu wykonującą blok `if` oraz drugą wykonującą blok `else`.

**5.5. Kolory obcych, część 3.** Przygotowaną w ćwiczeniu 5.4 konstrukcję `if-else` zastąp konstrukcją `if-elif-else`.

- Jeżeli kolor obcego to zielony, wyświetl komunikat informujący gracza, że zarobił pięć punktów za zestrzelenie tego obcego.
- Jeżeli kolor obcego to żółty, wyświetl komunikat informujący gracza, że zarobił dziesięć punktów za zestrzelenie tego obcego.
- Jeżeli kolor obcego to czerwony, wyświetl komunikat informujący gracza, że zarobił piętnaście punktów za zestrzelenie tego obcego.

- Przygotuj trzy wersje programu i upewnij się, że każdy komunikat zostanie wyświetlony dla odpowiedniego koloru zestrzelonego obcego.

**5.6. Etapy życia.** Przygotuj konstrukcję `if-else` ustalającą etap życia danej osoby. Przypisz wartość zmiennej `age`, a następnie:

- Jeżeli osoba ma mniej niż 2 lata, wyświetl komunikat informujący, że jest niemowlęciem.
- Jeżeli osoba ma co najmniej 2 lata, ale mniej niż 4, wyświetl komunikat informujący, że jest dzieckiem, które uczy się chodzić.
- Jeżeli osoba ma co najmniej 4 lata, ale mniej niż 13, wyświetl komunikat informujący, że jest dzieckiem.
- Jeżeli osoba ma co najmniej 13 lat, ale mniej niż 20, wyświetl komunikat informujący, że jest nastolakiem.
- Jeżeli osoba ma co najmniej 20 lat, ale mniej niż 65, wyświetl komunikat informujący, że jest dorosłym.
- Jeżeli osoba ma co najmniej 65 lat, wyświetl komunikat informujący, że jest seniorem.

**5.7. Ulubione owoce.** Przygotuj listę ulubionych owoców. Następnie utwórz serię niezależnych poleceń `if` sprawdzających obecność określonych owoców na liście.

- Utwórz listę o nazwie `favorite_fruits` i umieść na niej trzy ulubione owoce.
- Utwórz pięć poleceń `if`. Każde powinno sprawdzać obecność na liście określonego rodzaju owocu. Jeżeli owoc znajduje się na liście, w bloku `if` należy wyświetlić komunikat w stylu: „Naprawdę lubisz banany!”.

## Używanie poleceń `if` z listami

Dzięki połączeniu list i poleceń `if` można osiągnąć naprawdę interesujące efekty. Przykładem może być monitorowanie listy pod względem wartości specjalnych, które muszą być potraktowane w zupełnie inny sposób niż pozostałe wartości znajdujące się na liście. Dzięki temu zyskujemy możliwość efektywnego zarządzania zmieniającymi się wymaganiami, takimi jak dostępność określonych potraw w restauracji. Wspomniane połączenie list i poleceń `if` pozwala zagwarantować, że tworzony przez Ciebie kod we wszystkich możliwych sytuacjach będzie działał zgodnie z oczekiwaniemi.

### Sprawdzanie pod kątem wartości specjalnych

Ten rozdział rozpoczęliśmy od prostego przykładu pokazującego, jak obsłużyć wartość specjalną taką jak `'bmw'`, która musiała zostać wyświetlona w innym formacie niż wszystkie pozostałe wartości z listy. Skoro już poznajeś podstawy testów

warunkowych i poleceń `if`, możemy powrócić do tematu monitorowania listy pod kątem wartości specjalnych. Powiemy też, jak należy obsługiwać takie wartości.

Kontynuujemy omawianie przykładu pizzerii. Przygotowany wcześniej program wyświetla komunikat dotyczący dodatkowych składników pizzy oraz informuje, że pizza jest już gotowa. Kod tego rodzaju akcji można utworzyć bardzo efektywnie, umieszczając na liście dodatki wybrane przez klienta, a następnie przetwarzając tę listę za pomocą pętli. Podczas wykonywania pętli będzie wyświetlany komunikat wskazujący na aktualnie dodawany do pizzy składnik.

#### Plik toppings.py:

---

```
requested_toppings = ['pieczarki', 'boczek', 'podwójny ser']

for requested_topping in requested_toppings:
    print(f"Dodaję {requested_topping}.")

print("\nTwoja pizza jest już gotowa!")
```

---

Wygenerowane dane wyjściowe są proste, ponieważ sedno tego kodu to najprostsza pętla `for`:

---

```
Dodaję pieczarki.
Dodaję boczek.
Dodaję podwójny ser.
```

---

```
Twoja pizza jest już gotowa!
```

---

Jednak co zrobić, gdy w pizzerii zabraknie boczku? Umieszczenie polecenia `if` wewnętrz pętli `for` pozwala na odpowiednie obsłużenie tego rodzaju sytuacji:

---

```
requested_toppings = ['pieczarki', 'boczek', 'podwójny ser']

for requested_topping in requested_toppings:
    if requested_topping == 'boczek':
        print("Przepraszamy, ale obecnie nie mamy boczku.")
    else:
        print(f"Dodaję {requested_topping}.")

print("\nTwoja pizza jest już gotowa!")
```

---

Tym razem sprawdzamy każdy dodatkowy składnik przed jego dodaniem do pizzy. Polecenie `if` sprawdza, czy klient zażyczył sobie boczku. Jeżeli tak, zostaje wyświetlony komunikat o niedostępności tego dodatku. Z kolei blok `else` gwarantuje dodanie do pizzy wszystkich pozostałych składników, których zażyczył sobie klient.

Wygenerowane dane wyjściowe pokazują, że wszystkie podane przez klienta dodatki zostały prawidłowo obsłużone:

---

Dodaję pieczarki.  
Przepraszamy, ale obecnie nie mamy boczku.  
Dodaję podwójny ser.

Twoja pizza jest już gotowa!

---

## Sprawdzanie, czy lista nie jest pusta

Przyjęliśmy proste założenie, że każda lista, z którą dotąd pracowaliśmy zawiera przynajmniej jeden element. Wkrótce pozwolimy użytkownikom na umieszczanie informacji na liście, a tym samym nie będziemy mogli dłużej zakładać, że lista przetwarzana w pętli ma jakiekolwiek elementy. W takiej sytuacji przed przejściem do pętli `for` użyteczne będzie sprawdzenie, czy lista nie jest pusta.

Przykładowo w omawianym programie dla pizzerii, zanim rozpoczęniemy przygotowywanie pizzy, sprawdzamy, czy lista podana przez klienta zawiera przynajmniej jeden dodatek. Jeżeli lista jest pusta, wyświetlamy komunikat z pytaniem, czy na pewno chce zamówić pizzę bez dodatków. Natomiast jeśli lista zawiera co najmniej jeden dodatek, wtedy zaczynamy przygotowywać pizzę (podobnie jak we wcześniejszych przykładach):

---

```
requested_toppings = []

if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Dodaję {requested_topping}.")
        print("\nTwoja pizza jest już gotowa!")
else:
    print("Czy jesteś pewien, że chcesz pizzę bez dodatków?")
```

---

Tym razem zaczynamy od pustej listy dodatków wybranych przez klienta. Zamiast przejść od razu do pętli `for`, najpierw przeprowadzamy szybkie sprawdzenie. Kiedy nazwa listy pojawia się w poleceniu `if`, Python zwraca wartość `True`, jeśli lista zawiera przynajmniej jeden element. W przypadku pustej listy wartością zwrotną będzie `False`. Gdy lista `requested_toppings` przejdzie test warunkowy, wykonujemy tę samą pętlę, z której korzystaliśmy w poprzednich przykładach. Natomiast jeśli lista nie zaliczy testu warunkowego, klientowi wyświetlany jest komunikat z pytaniem, czy naprawdę chce zamówić pizzę bez żadnych dodatków.

W omawianym przykładzie lista jest pusta, więc wygenerowane dane wyjściowe to wspomniane pytanie o zrobienie pizzy bez dodatków:

---

Czy jesteś pewien, że chcesz pizzę bez dodatków?

---

Jeżeli lista nie będzie pusta, wygenerowane dane wyjściowe będą pokazywały dodawanie do zamówionej pizzy kolejnych wybranych przez klienta składników.

## Użycie wielu list

Klienci mogą zażyczyć sobie wszystkiego, zwłaszcza gdy chodzi o dodatki do pizzy. Co zrobić w sytuacji, gdy klient będzie chciał zamówić pizzę z frytkami? Zanim rozpoczęniemy przetwarzanie danych, możemy wykorzystać listy i polecenia if, aby się upewnić, że dane wejściowe mają sens.

Wzbogacimy teraz program o monitorowanie nietypowych dodatków, abyśmy wiedzieli o nich, zanim zaczniemy przygotowywać pizzę. Przedstawiony poniżej program zawiera dwie listy. Pierwsza z nich przechowuje dostępne dodatki dla pizzy, natomiast druga dodatki zamówione przez klienta. Tym razem każdy składnik z listy `requested_toppings`, zanim zostanie dodany do pizzy, będzie sprawdzany na liście dostępnych dodatków:

---

```
available_toppings = ['pieczarki', 'oliwki', 'boczek',
                      'pepperoni', 'ananas', 'podwójny ser']

requested_toppings = ['pieczarki', 'frytki', 'podwójny ser'] ❶

for requested_topping in requested_toppings:
    if requested_topping in available_toppings: ❷
        print(f"Dodaję {requested_topping}.")
    else: ❸
        print(f"Przepraszamy, ale obecnie nie mamy dodatku {requested_topping}.")

print("\nTwoja pizza jest już gotowa!")
```

---

Rozpoczynamy od zdefiniowania listy dostępnych dodatków do pizzy. Zwróć uwagę, że możemy tutaj wykorzystać krotkę, jeśli pizzeria oferuje stały wybór dodatków. Następnie definiujemy listę dodatków zamówionych przez klienta. Na tej liście pojawił się nietypowy dodatek, czyli 'frytki' ❶. Dalej kod przeprowadza iterację przez dodatki wybrane przez klienta. Wewnątrz pętli najpierw sprawdzamy, czy konkretny dodatek znajduje się na liście dostępnych składników (patrz wiersz ❷). Jeżeli tak, dodajemy go do pizzy. Natomiast jeśli wybranego przez klienta składnika nie ma na liście dostępnych dodatków, wykonywany będzie blok `else` zdefiniowany w wierszu ❸. Polecenie `print()` w bloku `else` wyświetla klientowi komunikat o niedostępności wybranego dodatku.

Składnia przedstawionego kodu powoduje wygenerowanie jasnych i przejrzystych danych wyjściowych:

---

```
Dodaję pieczarki.
Przepraszamy, ale obecnie nie mamy dodatku frytki.
Dodaję podwójny ser.

Twoja pizza jest już gotowa!
```

---

W zaledwie kilku wierszach kodu całkiem efektywnie poradziliśmy sobie z sytuacją, z którą można spotkać się w rzeczywistości.

## ZRÓB TO SAM

**5.8. Witaj, administratorze.** Przygotuj listę przynajmniej pięciu nazw użytkowników, zawierającą między innymi nazwę 'admin'. Wyobraź sobie, że tworzysz kod odpowiedzialny za powitanie użytkownika po jego zalogowaniu się w witrynie internetowej. Przeprowadź iterację przez pętlę i dla każdego użytkownika wyświetl powitanie.

- Jeżeli nazwa użytkownika to 'admin', wyświetl powitanie specjalne w stylu: „Witaj, admin! Czy chcesz przejrzeć dzisiejszy raport?”.
- Dla pozostałych użytkowników wyświetl zwykłe powitanie w stylu: „Witaj, Eryk! Dziękujemy, że ponownie się zalogowałeś”.

**5.9. Brak użytkowników.** Do przygotowanego w poprzednim ćwiczeniu pliku *hello\_admin.py* dodaj polecenie `if` sprawdzające, czy lista użytkowników nie jest pusta.

- Jeżeli lista jest pusta, wyświetl komunikat w stylu: „Musimy znaleźć jakichś użytkowników!”.
- Usuń wszystkich użytkowników z listy i upewnij się, że powyższy komunikat został prawidłowo wyświetlony.

**5.10. Sprawdzanie nazw użytkowników.** Wykonaj poniższe działania w celu utworzenia programu symulującego sposób, w jaki witryna internetowa gwarantuje, że każdy użytkownik będzie miał unikatową nazwę.

- Utwórz listę o nazwie `current_users` i umieść na niej przynajmniej pięć nazw użytkowników.
- Utwórz kolejną listę o nazwie `new_users`. Upewnij się, że przynajmniej jedna nazwa użytkownika z nowej listy znajduje się także na liście `current_users`.
- Przeprowadź iterację przez nową listę, aby sprawdzić, czy dana nazwa użytkownika została już wcześniej użyta. Jeśli tak, wyświetl użytkownikowi komunikat o konieczności wyboru innej nazwy. Natomiast jeśli dana nazwa użytkownika nie została wcześniej użyta, wyświetl użytkownikowi komunikat informujący o możliwości jej wykorzystania.
- Upewnij się, że w trakcie porównywania nazw użytkowników wielkość liter nie ma znaczenia. Jeżeli wcześniej użyto nazwy 'Janek', nowy użytkownik nie może wybrać dla siebie nazwy 'JANEK'. (To wymaga utworzenia kopii listy `current_users` zawierającej zapisane małymi literami nazwy wszystkich użytkowników).

**5.11. Liczby porządkowe.** Liczby porządkowe wskazują położenie elementu na liście, na przykład pierwszy, drugi, trzeci. W języku angielskim większość tego rodzaju liczb kończy się na „`th`”, poza liczbami 1, 2 i 3.

- Utwórz listę przechowującą liczby od 1 do 9.
- Przeprowadź iterację przez listę.
- Wykorzystaj konstrukcję `if-elif-else` wewnątrz pętli do prawidłowego wyświetlenia liczb porządkowych w języku angielskim. Wygenerowane dane wyjściowe powinny mieć postać 1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 8th, 9th, a każda liczba powinna znajdować się w osobnym wierszu.

## Nadawanie stylu poleceniom if

W każdym przykładzie przedstawionym w tym rozdziale mogłeś zobaczyć dobre nawyki dotyczące stosowanego stylu zapisywania kodu. Jedyna zamieszczona w PEP 8 rekomendacja związana z formatowania testów warunkowych dotyczy użycia pojedynczej spacji wokół operatorów porównania, takich jak `==`, `>=`, `<=`. Na przykład poniższy zapis:

---

```
if age < 4:
```

---

jest lepszy od następującego:

---

```
if age<4:
```

---

Takie pojedyncze spacje nie mają wpływu na sposób, w jaki Python interpretuje kod, a jedynie ułatwiają programistom jego odczytywanie.

### ZRÓB TO SAM

**5.12. Nadanie stylu konstrukcji if.** Przejrzyj programy utworzone w tym rozdziale i upewnij się, że zastosowałaś w nich odpowiednie formatowanie testów warunkowych.

**5.13. Twoje pomysły.** Na tym etapie masz już znacznie większe możliwości w zakresie programowania niż na początku książki. Powinieneś już wyraźniej dostrzegać, jak rzeczywiste sytuacje mogą być modelowane w programach. Możesz więc zastanowić się nad problemami, które chciałbyś rozwiązywać we własnych programach. Zanotuj nowe idee dotyczące problemów, które chciałbyś rozwiązać, gdy zdobędziesz jeszcze większe umiejętności w zakresie programowania. Zastanów się nad grami, które chciałbyś stworzyć, zbiorami danych, które chciałbyś przeanalizować, lub aplikacjami internetowymi, które chciałbyś zbudować.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak tworzyć testy warunkowe, które zawsze będą zwracały wartość `True` lub `False`. Zobaczyłeś, jak tworzyć proste polecenia `if`, a także konstrukcje `if-else` i `if-elif-else`. Rozpocząłeś wykorzystywanie wymienionych struktur do identyfikacji określonych sytuacji oraz sprawdzenia, kiedy występują one w programach. Nauczyłeś się obsługiwać specjalne elementy listy w zupełnie inny sposób niż pozostałe, jednocześnie wykorzystując efektywność oferowaną przez pętlę `for`. Ponadto poznałeś zalecenia dotyczące stylu tworzenia kodu. Dzięki zastosowaniu odpowiedniego stylu coraz bardziej skomplikowane programy wciąż pozostaną względnie łatwe do odczytania i zrozumienia.

W rozdziale 6. poznasz słowniki Pythona. Są one podobne do list, ale pozwalały na łączenie fragmentów informacji. Dowiesz się, jak budować takie słowniki, przeprowadzać przez nie iteracje oraz wykorzystywać je w połączeniu z listami i poleceniami `if`. Dzięki opanowaniu słowników będziesz mógł modelować jeszcze więcej rzeczywistych sytuacji.

# 6

## Słowniki



W TYM ROZDZIALE DOWIESZ SIĘ, JAK UŻYWAĆ W PYTHONIE SŁOWNIKÓW, KTÓRE POZWALAJĄ POŁĄCZYĆ POWIĄZANE ZE SOBĄ INFORMACJE. ZOBACZYSZ, JAK UZYSKAĆ DOSTĘP DO DANYCH ZNAJDUJĄCYCH SIĘ W SŁOWNIKU oraz jak modyfikować te dane. Ponieważ słowniki mogą przechowywać praktycznie nieograniczoną ilość informacji, zaprezentuję iterację przez dane umieszczone w słowniku. Ponadto nauczysz się zagnieźdzać słowniki wewnętrz list, listy wewnętrz słowników, a nawet słowniki wewnętrz innych słowników.

Poznanie słowników pozwoli Ci znacznie wierniej modelować różne rzeczywiste obiekty. Zyskasz możliwość utworzenia słownika przedstawiającego osobę i przechowującą wszystkie informacje o tej osobie. Będziesz mógł na przykład przechowywać takie dane jak imię i nazwisko, wiek, miejsce zamieszkania, zawód, a także wszelkie inne dane opisujące tę osobę. Ponadto będziesz mógł przechowywać dwa dowolne rodzaje informacji, które będą do siebie dopasowane, na przykład listę słów i ich znaczenie, listę osób i ich ulubione liczby, listę szczytów i ich wysokości.

### Prosty słownik

Rozważ grę, w której występują obcy o różnych kolorach, a liczba punktów używanych po zestrzeleniu obcego jest zależna od jego koloru. Poniżej przedstawiłem słownik przeznaczony do przechowywania informacji o obcym.

*Plik alien.py:*

---

```
alien_0 = {'color': 'zielony', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

---

W słowniku `alien_0` przechowujemy kolor obcego oraz liczbę punktów otrzymywanych za jego unicestwienie. Dwa wywołania `print()` uzyskują dostęp do słownika i wyświetlają przechowywane w nim informacje:

---

```
zielony
5
```

---

Podobnie jak to jest w przypadku większości nowych koncepcji w programowaniu, przywyknięcie do słowników wymaga praktyki. Kiedy nabędziesz nieco doświadczenia w pracy ze słownikami, przekonasz się, jak można efektywnie wykorzystywać je do modelowania rzeczywistych sytuacji.

## Praca ze słownikami

W Pythonie *słownik* jest kolekcją *par klucz-wartość*. Każdy *klucz* jest połączony z wartością, za pomocą klucza można uzyskać dostęp do powiązanej z nim wartości. Wartością klucza może być liczba, ciąg tekstowy, lista, lub nawet inny słownik. W rzeczywistości wartością słownika może być dowolny obiekt możliwy do utworzenia w Pythonie.

Słownik w Pythonie jest opakowany w nawiasy klamrowe i zawiera serię par klucz-wartość, tak jak pokazałem w poprzednim przykładzie:

---

```
alien_0 = {'color': 'zielony', 'points': 5}
```

---

*Para klucz-wartość* to zbiór wartości powiązanych ze sobą. Kiedy podajesz klucz, Python zwraca powiązaną z nim wartość. Połączenie klucza z wartością odbywa się za pomocą dwukropka, a poszczególne pary klucz-wartość są rozdzielone przecinkami. W słowniku można przechowywać dowolną liczbę par klucz-wartość.

Najprostszy słownik ma dokładnie jedną parę klucz-wartość, tak jak pokazałem poniżej w zmodyfikowanej wersji słownika `alien_0`:

---

```
alien_0 = {'color': 'zielony'}
```

---

Ten słownik przechowuje jeden fragment informacji dotyczący obcego, a dokładnie jego kolor. W omawianym słowniku ciąg tekstowy 'color' jest kluczem, z którym jest powiązana wartość 'zielony'.

## Uzyskiwanie dostępu do wartości słownika

Aby pobrać wartość powiązaną z kluczem, należy podać nazwę słownika oraz nazwę klucza ujętą w nawias kwadratowy:

```
alien_0 = {'color': 'zielony'}  
print(alien_0['color'])
```

Wywołanie `print()` wyświetla wartość klucza 'color' w słowniku `alien_0`:

```
zielony
```

Słownik może zawierać nieograniczoną liczbę par klucz-wartość. Przykładowo poniżej znajduje się początkowy słownik `alien_0` z dwiema parami klucz-wartość:

```
alien_0 = {'color': 'zielony', 'points': 5}
```

Teraz można uzyskać dostęp do koloru obcego oraz liczby punktów przyznawanych za jego zestrzelenie. Jeżeli gracz zestrzeli obcego, liczbę punktów, które należy mu przyznać, można sprawdzić za pomocą kodu podobnego do poniższego:

```
alien_0 = {'color': 'zielony', 'points': 5}  
  
new_points = alien_0['points']  
print(f"Zdobyłeś {new_points} punktów!")
```

Odwołując się do zdefiniowanego słownika, kod pobiera wartość przypisaną do klucza 'points'. Następnie ta wartość zostaje umieszczona w zmiennej `new_points`. Kod w ostatnim wierszu konwertuje liczbę całkowitą do postaci ciągu tekstopowego i wyświetla komunikat o liczbie punktów zdobytych przez gracza:

```
Zdobyłeś 5 punktów!
```

Jeżeli powyższy kod będzie wykonywany po każdym zestrzeleniu obcego, będziesz mógł pobierać liczbę punktów przyznawanych za unicestwienie danego obcego.

## Dodanie nowej pary klucz-wartość

Słownik to struktura dynamiczna, więc nowe pary klucz-wartość można dodawać w każdej chwili. W celu dodania nowej pary klucz-wartość należy podać nazwę słownika, ujętą w nawias kwadratowy nazwę nowego klucza oraz wartość przypisywaną do danego klucza.

Do przedstawionego wcześniej słownika alien\_0 dodamy teraz dwie nowe informacje: współrzędne X i Y położenia obcego, co ułatwi nam jego wyświetlenie w odpowiednim miejscu na ekranie. Umieścimy teraz obcego przy lewej krawędzi, w odległości 25 pikseli od górnej krawędzi ekranu. Ponieważ współrzędne ekranu zwykle rozpoczynają się w lewym górnym rogu, w celu umieszczenia obcego przy lewej krawędzi należy współrzędnej X przypisać wartość 0, natomiast jego odsumięcie o 25 pikseli od górnej krawędzi ekranu wymaga przypisania współrzędnej Y wartości 25, tak jak przedstawiłem w poniższym fragmencie kodu:

*Plik alien.py:*

---

```
alien_0 = {'color': 'zielony', 'points': 5}
print(alien_0)

alien_0['x_position'] = 0
alien_0['y_position'] = 25
print(alien_0)
```

---

Rozpoczynamy od zdefiniowania takiego samego słownika jak wcześniej. Następnie wyświetlamy jego zawartość, aby w ten sposób mieć punkt odniesienia. Później dodajemy do słownika nową parę klucz-wartość: klucz 'x\_position', wartość 0. Do słownika dodajemy także drugą nową parę: tym razem klucz to 'y\_position', natomiast wartość to 25. Po ponownym wyświetleniu zmodyfikowanego słownika można dostrzec, że nowe pary klucz-wartość faktycznie zostały w nim umieszczone:

---

```
{'color': 'zielony', 'points': 5}
{'color': 'zielony', 'points': 5, 'y_position': 25, 'x_position': 0}
```

---

Ostateczna wersja słownika zawiera cztery pary klucz-wartość. Dwie początkowe określają kolor obcego i liczbę punktów przyznawanych za jego zestrzelenie. Z kolei dwie nowe pary przechowują informacje o położeniu obcego na ekranie.

Słownik zachowuje kolejność, w której były dodawane pary klucz-wartość. Podczas wyświetlania słownika lub iteracji przez jego elementy zostaną one wyświetlane w kolejności ich dodawania do słownika.

## Rozpoczęcie pracy od pustego słownika

Czasami będzie wygodne lub wręcz konieczne rozpoczęcie pracy od pustego słownika, do którego dopiero później będą wstawiane pary klucz-wartość. Aby rozpocząć wypełnianie pustego słownika, zdefiniuj go za pomocą pustego nawiasu klamrowego, a następnie dodawaj poszczególne pary klucz-wartość, po jednej w każdym wierszu. Poniżej pokazałem budowanie słownika `alien_0` z zastosowaniem tego rodzaju podejścia:

*Plik alien.py:*

---

```
alien_0 = {}

alien_0['color'] = 'zielony'
alien_0['points'] = 5

print(alien_0)
```

---

W powyższym fragmencie kodu zdefiniowaliśmy pusty słownik `alien_0`, do którego później dodaliśmy informacje o kolorze obcego i punktach przyznawanych za jego zestrzelenie. Wynikiem jest powstanie słownika dokładnie takiego samego jak we wcześniejszych przykładach:

---

```
{'color': 'zielony', 'points': 5}
```

---

Pusty słownik tworzymy najczęściej wtedy, kiedy chcemy przechowywać dane dostarczane przez użytkownika lub mamy do czynienia z kodem, który automatycznie generuje ogromną liczbę par klucz-wartość.

## Modyfikowanie wartości słownika

Aby zmodyfikować wartość w słowniku, należy podać jego nazwę, ujętą w nawias kwadratowy nazwę klucza oraz nową wartość, która ma zostać przypisana do wskazanego klucza. Rozważmy na przykład sytuację, gdy w trakcie gry obcy zmienia kolor z zielonego na żółty:

*Plik alien.py:*

---

```
alien_0 = {'color': 'zielony'}
print(f"Obcy ma kolor {alien_0['color']}.")

alien_0['color'] = 'żółty'
print(f"Obcy ma teraz kolor {alien_0['color']}.)")
```

---

Zaczynamy od zdefiniowania słownika `alien_0` zawierającego jedynie informację o kolorze obcego. Następnie wartość klucza 'color' zmieniamy z 'zielony' na 'żółty'. Wygenerowane dane wyjściowe pokazują, że kolor obcego faktycznie został zmieniony z zielonego na żółty:

---

Obcy ma kolor zielony.  
Obcy ma teraz kolor żółty.

---

Bardziej interesującym przykładem może być monitorowanie położenia obcego, który porusza się z różną szybkością. W słowniku przechowujemy wartość określającą bieżącą szybkość obcego i używamy jej do ustalenia odległości, jaką powinien pokonać obcy poruszający się w prawą stronę:

---

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'średnio'}
print(f"Początkowa wartość x-position: {alien_0['x_position']}")

# Przesunięcie obcego w prawo.
# Ustalenie odległości, jaką powinien pokonać obcy poruszający się z daną szybkością.
if alien_0['speed'] == 'wolno': ❶
    x_increment = 1
elif alien_0['speed'] == 'średnio':
    x_increment = 2
else:
    # To musi być szybki obcy.
    x_increment = 3

# Nowe położenie to suma doteczczasowego położenia i wartości x_increment.
alien_0['x_position'] = alien_0['x_position'] + x_increment ❷

print(f"Nowa wartość x-position: {alien_0['x_position']}")
```

---

Rozpoczynamy od zdefiniowania obcego wraz z początkowym położeniem X i Y, a także szybkością określoną jako 'średnio'. W celu zachowania prostoty przykładu pomijamy wartości koloru i przyznawanych punktów, ale ten przykład oczywiście będzie wyglądał dokładnie tak samo po uwzględnieniu wspomnianych danych. Wyświetlamy pierwotną wartość x\_position, aby pokazać, o jaką odległość w prawą stronę przesunął się obcy.

W wierszu ❶ konstrukcja if-elif-else pozwala ustalić odległość, jaką powinien pokonać obcy przesuwający się w prawą stronę, i przechowuje ją w zmiennej x\_increment. Jeżeli szybkość obcego jest określona jako 'wolno', przesunie się on tylko o jedną jednostkę w prawo. Szybkość 'średnio' powoduje przesunięcie się o dwie jednostki w prawo, natomiast 'szybko' o trzy. Kiedy zostanie ustalona odległość do przebycia, w wierszu ❷ do aktualnej wartości klucza x\_position dodajemy wartość zmiennej x\_increment, a sumę umieszczymy w kluczu x\_position słownika.

Ponieważ mamy do czynienia z obcym poruszającym się ze średnią szybkością, przesunie się on o dwie jednostki w prawo:

---

Początkowa wartość x-position: 0  
Nowa wartość x-position: 2

---

Ta technika jest całkiem dobra: dzięki zmianie jednej wartości w słowniku opisującym obcego możemy zmienić też jego ogólne zachowanie. Na przykład poniższe polecenie sprawia, że nasz średnio szybki obcy zaczyna poruszać się szybko:

---

```
alien_0['speed'] = szybko
```

---

W trakcie następnego wykonywania kodu blok konstrukcji `if-elif-else` przypisze zmiennej `x_increment` większą wartość.

## Usuwanie pary klucz-wartość

Kiedy przechowywany w słowniku fragment informacji nie jest dłużej potrzebny, za pomocą polecenia `del` można całkowicie usunąć parę klucz-wartość. Do prawidłowego działania polecenie `del` potrzebuje mieć podaną nazwę słownika oraz klucz przeznaczony do usunięcia.

Na przykład ze słownika `alien_0` chcemy usunąć klucz `'points'` wraz z jego wartością:

*Plik alien.py:*

---

```
alien_0 = {'color': 'zielony', 'points': 5}
print(alien_0)

del alien_0['points'] ❶
print(alien_0)
```

---

Polecenie w wierszu ❶ nakazuje Pythonowi usunięcie klucza `'points'` ze słownika `alien_0`, a także wartości powiązanej z wymienionym kluczem. Wygenerowane dane wyjściowe pokazują, że klucz `'points'` i jego wartość 5 zostały usunięte, natomiast pozostała część słownika nie została zmieniona:

---

```
{'color': 'zielony', 'points': 5}
{'color': 'zielony'}
```

---

**UWAGA** Należy pamiętać, że operacja usunięcia pary klucz-wartość jest nieodwracalna.

## Słownik podobnych obiektów

W poprzednim przykładzie przechowywaliśmy różne rodzaje informacji o jednym obiekcie, czyli o obcym w grze. Jednak słownik można wykorzystać także do przechowywania jednego rodzaju informacji o wielu obiektach. Na przykład przeprowadzamy ankietę dotyczącą ulubionego języka programowania. W takim przypadku słownik będzie użyteczną strukturą przeznaczoną do przechowywania wyników tak prostej ankiety:

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}
```

---

Jak możesz zobaczyć, definicja większego słownika została podzielona na kilka wierszy kodu. Każdy klucz to imię uczestnika ankiety, natomiast wartość to podany przez niego ulubiony język programowania. Kiedy wiesz, że do utworzenia słownika potrzebujesz więcej niż tylko jednego wiersza kodu, po nawiasie otwierającym naciśnij klawisz *Enter*. W ten sposób powstanie wcięcie o wielkości jednego poziomu (cztery spacje) i zostanie zapisana pierwsza para klucz-wartość wraz z przecinkiem. Od tego momentu kolejne naciśnięcia klawisza *Enter* powinny powodować, że edytor tekstu automatycznie będzie stosował wcięcia dla następnych par klucz-wartość, aby dopasować wielkość tych wcięć do pierwszej pary.

Gdy zakończysz definiowanie słownika, w nowym wierszu po ostatniej parze klucz-wartość umieść nawias zamkujący i zastosuj wcięcie na poziomie równym kluczom słownika. Dobrą praktyką jest umieszczenie przecinka także po ostatniej parze klucz-wartość, aby definicja słownika była gotowa na dodanie nowej pary klucz-wartość w następnym wierszu.

**UWAGA** Większość edytorów oferuje pewnego rodzaju funkcjonalność pomagającą w formatowaniu rozbudowanych list i słowników w sposób podobny do przedstawionego w przykładzie. Dostępne są jeszcze inne akceptowalne sposoby formatowania długich słowników, więc w używanym edytorze oraz w innych źródłach będziesz mógł się spotkać z nieco odmiennym formatowaniem.

Dzięki tak przygotowanemu słownikowi, znając imię uczestnika ankiety, możemy bardzo łatwo pobrać informacje o jego ulubionym języku programowania.

*Plik favorite\_languages.py:*

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}

language = favorite_languages['sara'].title() ❶
print(f"Ulubiony język programowania Sary to {language}.")
```

---

W celu wyświetlenia ulubionego języka programowania Sary należy użyć przedstawionego poniżej polecenia:

---

```
favorite_languages['sara']
```

---

Powyższa składnia została użyta do pobrania ze słownika ulubionego języka programowania Sary ❶ i przypisania go zmiennej `language`. Dzięki utworzeniu nowej zmiennej otrzymujemy znacznie czytelniejsze wywołanie `print()`. Wygenerowane dane wyjściowe zawierają informacje o ulubionym języku programowania Sary, tak jak pokazałem poniżej:

---

```
Ulubiony język programowania Sary to C.
```

---

Tę samą składnię można wykorzystać do przedstawienia dowolnego elementu ze słownika.

## Używanie metody `get()` w celu uzyskania dostępu do wartości

Używanie umieszczonego w nawiasach kwadratowych kluczy w celu pobierania żądanego wartości ze słownika może prowadzić do potencjalnego problemu: jeśli podany klucz nie istnieje, wynikiem będzie błąd.

Zobacz, co się stanie, gdy spróbujesz pobrać ze słownika nieistniejącą wartość:

*Plik alien\_no\_points.py:*

---

```
alien_0 = {'color': 'zielony', 'speed': 'wolno'}
print(alien_0['points'])
```

---

Wynikiem jest wyświetlenie stosu wywołań wskazującego na wystąpienie błędu typu `KeyError`:

---

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
                ~~~~~~^~~~~~
KeyError: 'points'
```

---

W rozdziale 10. znajdziesz więcej informacji na temat obsługi błędów takich jak w omawianym przykładzie. Podczas pracy ze słownikiem można wykorzystać metodę `get()` do zdefiniowania wartości domyślnej, która będzie zwrócona, jeśli żądany klucz nie istnieje.

Metoda `get()` wymaga podania klucza jako pierwszego argumentu. Drugim, opcjonalnym argumentem jest wartość zwracana w przypadku, gdy podany klucz nie istnieje w słowniku.

---

```
alien_0 = {'color': 'zielony', 'speed': 'wolno'}  
  
point_value = alien_0.get('points', 'Brak przypisanych punktów.')  
print(point_value)
```

---

Jeżeli klucz `'points'` istnieje w słowniku, otrzymasz przypisaną mu wartość. Natomiast jeśli klucz nie istnieje, otrzymasz wartość domyślną. W omawianym przykładzie klucz `'points'` nie istnieje w słowniku, więc zamiast błędu otrzymujesz czytelny komunikat:

---

```
Brak przypisanych punktów.
```

---

Jeżeli istnieje niebezpieczeństwo, że żądany klucz nie znajduje się w słowniku, rozważ wykorzystanie metody `get()` zamiast składni z użyciem nawiasu kwadratowego.

**UWAGA** Jeżeli pominiesz drugi argument w wywołaniu `get()`, a klucz nie istnieje, wówczas Python zwróci wartość `None`. Jest to wartość specjalna oznaczająca „brak wartości”. To nie jest błąd, lecz jedynie wartość specjalna wskazująca na brak wyraźnie zdefiniowanej wartości. Więcej przykładów użycia `None` przedstawię w rozdziale 8.

## ZRÓB TO SAM

**6.1. Osoba.** Wykorzystaj słownik do przechowywania informacji o znanej Ci osobie. W słowniku powinny znaleźć się informacje takie jak imię, nazwisko, wiek i miasto zamieszkania. Powinieneś więc utworzyć klucze `first_name`, `last_name`, `age` i `city`. Następnie wyświetl wszystkie informacje przechowywane w słowniku.

**6.2. Ulubione liczby.** Wykorzystaj słownik do przechowywania ulubionych liczb różnych osób. Weź pod uwagę pięć osób i ich imion użyj w charakterze kluczy słownika. Następnie ustal ich ulubione liczby i umieść je w słowniku, przypisując każdej osobie po jednej liczbie. Wyświetl imiona wszystkich osób i ich ulubione liczby. Jeżeli chcesz mieć więcej frajd podczas wykonywania tego ćwiczenia, zapytaj przyjaciół o ich ulubione liczby i umieść w programie rzeczywiste dane.

**6.3. Glosariusz.** Słownik Pythona można wykorzystać do przygotowania rzeczywistego słownika. Jednak w celu uniknięcia niejasności nazwiemy go glosariuszem.

- Wypisz sobie pięć słów z dziedziny programowania, które poznaleś we wcześniejszych rozdziałach. Te słowa będą kluczami w glosariuszu, natomiast wartościami będą znaczenia poszczególnych słów.
- Każde słowo i jego znaczenie wyświetl w postaci elegancko sformatowanych danych wyjściowych. Możesz w jednym wierszu wyświetlić słowo, dwukropek i później wyjaśnienie danego słowa. Eventualnie słowo umieść w jednym wierszu, a jego wyjaśnienie w kolejnym, wciętym wierszu. Do wstawienia pustego wiersza między parami słowo-definicja użyj znaku nowego wiersza (\n).

## Iteracja przez słownik

Pojedynczy słownik Pythona może zawierać od kilku do nawet kilku milionów par klucz-wartość. Ponieważ w słowniku może znaleźć się ogromna ilość danych, Python pozwala przeprowadzać iterację przez słownik. Ponadto słowniki mogą być wykorzystywane do przechowywania informacji na wiele różnych sposobów, więc istnieje kilka odmiennych rozwiązań w zakresie iteracji przez słownik. Możliwa jest iteracja przez wszystkie pary klucz-wartość słownika albo tylko przez jego klucze lub wartości.

### Iteracja przez wszystkie pary klucz-wartość

Zanim przejdziemy do omawiania różnych podejść w zakresie iteracji, najpierw spójrz na nowy słownik przeznaczony do przechowywania informacji o użytkowniku witryny internetowej. Przedstawiony poniżej słownik zawiera nazwę użytkownika, imię oraz nazwisko jednej osoby:

Plik user.py:

---

```
user_0 = {
    'username': 'jkowalski',
    'first': 'jan',
    'last': 'kowalski',
}
```

---

Na podstawie wiedzy zdobytej dotąd w tym rozdziale potrafisz uzyskać dostęp do pojedynczej informacji dotyczącej użytkownika, którego dane znajdują się w słowniku user\_0. Co możesz zrobić w sytuacji, gdy ze słownika chcesz pobrać wszystkie informacje o danym użytkowniku? W tym celu za pomocą pętli for możesz przeprowadzić iterację przez słownik.

---

```
user_0 = {
    'username': 'jkowalski',
    'first': 'jan',
    'last': 'kowalski',
}
```

---

```
for key, value in user_0.items():
    print(f"\nKlucz: {key}")
    print(f"Wartość: {value}")
```

---

W celu przygotowania pętli `for` dla słownika konieczne jest utworzenie dwóch zmiennych przechowujących klucz i wartość każdej pary klucz-wartość. Dla wspomnianych zmiennych możesz wybrać dowolne nazwy. Powyższy kod będzie również działał bez problemów, jeśli dla nazw zmiennych użyjesz skrótów, na przykład:

```
for k, v in user_0.items()
```

---

W części drugiej polecenia `for` mamy nazwę słownika oraz metodę `items()`, której wartością zwrotną jest lista par klucz-wartość. Następnie pętla `for` przechowuje poszczególne pary w dwóch przedstawionych zmiennych. W omawianym fragmencie kodu zmienne wykorzystujemy do wyświetlenia klucza oraz przypisanej mu wartości. Sekwencja `\n` w pierwszym poleceniu `print()` zapewnia wstawienie w wygenerowanych danych wyjściowych pustego wiersza przed każdą parą klucz-wartość:

---

```
Klucz: username
Wartość: jkowalski

Klucz: first
Wartość: jan

Klucz: last
Wartość: kowalski
```

---

Iteracja przez wszystkie pary klucz-wartość sprawdza się wyjątkowo dobrze w przypadku słowników takich jak ten użyty w przedstawionym wcześniej programie `favorite_languages.py`, który przechowuje ten sam rodzaj informacji dla wielu różnych kluczy. Jeżeli przeprowadzisz iterację przez słownik `favorite_languages`, dane wyjściowe będą zawierały imię każdej osoby w słowniku oraz jej ulubiony język programowania. Ponieważ klucze zawsze odwołują się do imienia osoby, a wartość zawsze przedstawia język programowania, więc zmiennym w pętli `for` można nadać nazwy `name` i `language` zamiast ogólnych `key` i `value`. To znacznie ułatwi zrozumienie przeznaczenia danej pętli.

*Plik favorite\_languages.py:*

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}
```

```
for name, language in favorite_languages.items():
    print(f"Ulubiony język programowania użytkownika {name.title()} to
{language.title()}.")
```

---

W kodzie nakazujemy Pythonowi przeprowadzenie iteracji przez wszystkie pary klucz-wartość w słowniku. Podczas tej operacji klucz każdej pary jest przechowywany w zmiennej `name`, natomiast jego wartość w zmiennej `language`. Tego rodzaju jasne i czytelne nazwy znacznie ułatwiają pokazanie, na czym polega działanie wywołania `print()`.

W ten sposób za pomocą zaledwie kilku wierszy kodu można wyświetlić wszystkie informacje otrzymane od uczestników ankiety:

---

```
Ulubiony język programowania użytkownika Janek to Python.
Ulubiony język programowania użytkownika Sara to C.
Ulubiony język programowania użytkownika Paweł to Python.
Ulubiony język programowania użytkownika Edward to Rust.
```

---

Taki rodzaj pętli będzie się sprawdzał równie dobrze, gdy słownik będzie zawierał wyniki ankiety, w której wzięło udział na przykład milion osób.

## Iteracja przez wszystkie klucze słownika

Metoda `keys()` jest użyteczna, gdy nie trzeba przetwarzać wszystkich wartości znajdujących się w słowniku. W poniższym fragmencie kodu przeprowadzamy iterację przez słownik `favorite_languages` i wyświetlamy imiona wszystkich uczestników ankiety:

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}

for name in favorite_languages.keys():
    print(name.title())
```

---

W pętli `for` nakazujemy Pythonowi pobranie wszystkich kluczy ze słownika `favorite_languages` i przechowujemy je pojedynczo w zmiennej `name`. Wygenerowane dane wyjściowe zawierają imiona wszystkich uczestników ankiety:

---

```
Janek
Sara
Paweł
Edward
```

---

Iteracja przez klucze to tak naprawdę zachowanie domyślne podczas iteracji przez słownik, więc te same dane wyjściowe otrzymasz też po użyciu polecenia:

---

```
for name in favorite_languages:
```

---

zamiast:

---

```
for name in favorite_languages.keys():
```

---

Możesz zdecydować się na wyraźne użycie metody `keys()`, jeśli dzięki temu kod będzie łatwiejszy do odczytania, lub zupełnie ją pominąć.

Wewnątrz pętli `for` możesz uzyskać dostęp do wartości powiązanej z kluczem, co wymaga użycia bieżącego klucza. Kilku przyjaciolom wyświetlimy teraz komunikat o wybranych przez nich językach programowania. Podobnie jak to robiliśmy wcześniej, przeprowadzamy iterację przez imiona w słowniku, ale kiedy dopasujemy imię do jednego z naszych przyjaciół, wyświetlamy komunikat dotyczący jego ulubionego języka programowania:

---

```
favorite_languages = {  
    --ciecje--  
}  
  
friends = ['paweł', 'sara']  
for name in favorite_languages.keys():  
    print(f"Witaj, {name.title()}")  
  
if name in friends: ❶  
    language = favorite_languages[name].title() ❷  
    print(f"\tWitaj, {name.title()}! Widzę, że Twoim ulubionym  
    językiem programowania jest {language}!")
```

---

Rozpoczynamy od utworzenia listy przyjaciół, którym chcemy wyświetlić komunikat. Wewnątrz pętli wyświetlamy imiona wszystkich osób. Następnie wierszu ❶ sprawdzamy, czy imię aktualnie przechowywane w zmiennej `name` odpowiada imieniu znajdującemu się na liście `friends`. Jeżeli tak, dostęp do ulubionego języka programowania, używamy nazwy słownika oraz bieżącej wartości `name` jako klucza (patrz wiersz ❷). Następnie wyświetlamy specjalne powitanie odwołujące się do ulubionego języka programowania.

Wygenerowane dane wyjściowe zawierają imiona wszystkich osób, natomiast nasi przyjaciele otrzymują jeszcze specjalny komunikat:

---

Witaj, Edward.

Witaj, Paweł.

Witaj, Paweł! Widzę, że Twoim ulubionym językiem programowania jest Python!

Witaj, Sara.

Witaj, Sara! Widzę, że Twoim ulubionym językiem programowania jest C!  
Witaj, Janek.

---

Możliwe jest również użycie metody `keys()` do odszukania określonej osoby, która wzięła udział w ankcie. Tym razem sprawdzamy, czy Elżbieta wzięła udział w ankcie:

---

```
favorite_languages = {
    --cięcie--
}

if 'elżbieta' not in favorite_languages.keys():
    print("Elżbieta, proszę, weź udział w naszej ankiecie!")
```

---

Metoda `keys()` nie służy jedynie do przeprowadzania iteracji. W rzeczywistości zwraca listę wszystkich kluczy, a polecenie `if` po prostu sprawdza, czy '`elżbieta`' znajduje się na liście. Ponieważ Elżbiety nie ma na liście, zachęcamy ją do wzięcia udziału w ankcie:

---

Elżbieta, proszę, weź udział w naszej ankiecie!

---

## Iteracja przez uporządkowane klucze słownika

Iteracja przez słownik zawsze zwraca elementy w kolejności ich wstawiania do słownika. Jednak czasami zachodzi potrzeba przeprowadzenia iteracji w zupełnie innej kolejności.

Jednym sposobem, aby elementy zostały zwrócone w określonej kolejności, jest posortowanie kluczy po ich otrzymaniu w pętli `for`. Funkcję `sorted()` możemy wykorzystać do uzyskania uporządkowanych kopii kluczy:

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, dziękujemy za udział w ankiecie.")
```

---

Powyższe polecenie `for` jest podobne do wcześniejszych, z wyjątkiem tego, że wywołanie metody `dictionary.keys()` zostało opakowane funkcją `sorted()`. W ten sposób nakazaliśmy Pythonowi wyświetlenie wszystkich kluczy słownika oraz posortowanie listy przed przeprowadzeniem iteracji. Wygenerowane dane wyjściowe pokazują, że imiona uczestników ankiety zostały wyświetlone w kolejności alfabetycznej:

---

Edward, dziękujemy za udział w ankcie.  
Janek, dziękujemy za udział w ankcie.  
Paweł, dziękujemy za udział w ankcie.  
Sara, dziękujemy za udział w ankcie.

---

## Iteracja przez wszystkie wartości słownika

Jeżeli interesują nas przede wszystkim wartości przechowywane w słowniku, wówczas można wykorzystać metodę `values()` w celu zwrotu listy wartości bez jakichkolwiek kluczy. Przykładowo przyjmujemy założenie, że chcemy pobrać listę wszystkich języków programowania wymienionych przez uczestników ankiety i nie interesują nas imiona osób wskazujących poszczególne języki:

---

```
favorite_languages = {
    'janek': 'python',
    'sara': 'c',
    'edward': 'rust',
    'paweł': 'python',
}

print("W ankcie zostały wymienione następujące języki programowania:")
for language in favorite_languages.values():
    print(language.title())
```

---

Powyższe pętla `for` pobiera wszystkie wartości ze słownika i przechowuje je w zmiennej `language`. Po wyświetleniu poszczególnych wartości otrzymujemy listę wszystkich języków programowania wymienionych przez uczestników ankiety:

---

```
W ankcie zostały wymienione następujące języki programowania:
Python
C
Rust
Python
```

---

Tego rodzaju podejście pobiera wszystkie wartości ze słownika bez sprawdzania, czy się powtarzają. Przedstawione rozwiązanie może być wystarczające dla małej liczby wartości, ale w przypadku ankiety przeprowadzanej na ogromnej liczbie respondentów otrzymamy listę z wieloma powtarzającymi się wartościami. Aby wyświetlić jedynie unikatowe wartości, możemy użyć zbioru. Wspomniany *zbiór* jest podobny do listy, ale każdy znajdujący się w nim element musi być unikatowy:

---

```
favorite_languages = {
    --cięcie--
}
```

```
print("W ankiecie zostały wymienione następujące języki programowania:")
for language in set(favorite_languages.values()):
    print(language.title())
```

---

Kiedy lista zawierająca powielające się elementy jest opakowana wywołaniem `set()`, Python identyfikuje wszystkie unikatowe elementy na liście, a następnie na ich podstawie tworzy zbiór. W omawianym kodzie wykorzystujemy wywołanie `set()` do pobrania unikatowych nazw języków otrzymanych jako wynik działania metody `favorite_languages.values()`.

Wynikiem jest lista języków wymienionych przez uczestników ankiety, która nie zawiera powtarzających się elementów:

---

```
W ankiecie zostały wymienione następujące języki programowania:
Python
C
Rust
```

---

Gdy będziesz kontynuować poznawanie Pythona, bardzo często odkryjesz wbudowane funkcje języka pomagające w przetworzeniu danych dokładnie w oczekiwany przez Ciebie sposób.

**UWAGA** *Zbiór można utworzyć bezpośrednio za pomocą nawiasu klamrowego, w którym trzeba umieścić rozdzielone przecinkami elementy:*

---

```
>>> languages = {'python', 'rust', 'python', 'c'}
>>> languages
{'rust', 'python', 'c'}
```

---

*Bardzo łatwo pomylić zbiór ze słownikiem, ponieważ w obu przypadkach stosowany jest nawias klamrowy. Gdy widzisz nawias klamrowy, ale bez par klucz-wartość, prawdopodobnie masz do czynienia ze zbiorem. W przeciwnieństwie do list i słowników zbiór nie przechowuje elementów w żadnej konkretnej kolejności.*

## ZRÓB TO SAM

**6.4. Glosariusz 2.** Skoro już wiesz, jak można przeprowadzić iterację przez słownik, to zmodyfikuj kod ćwiczenia 6.3 z wcześniejszej części rozdziału. Zastąp serię wywołań `print()` pętlą przeprowadzającą iterację przez klucze i wartości słownika. Po upewnieniu się, że pętla działa prawidłowo, do glosariusza dodaj kolejnych pięć terminów związanych z Pythonem. Kiedy ponownie uruchomisz program, nowo dodane terminy i ich definicje powinny zostać automatycznie uwzględnione w wyświetlonych danych wyjściowych.

**6.5. Rzeki.** Utwórz słownik zawierający trzy ważne rzeki oraz kraje, przez które one płyną. Jedna z par klucz-wartość może mieć postać 'nil': 'egipt'.

- Wykorzystaj pętlę do wyświetlenia zdania o każdej rzece, na przykład:  
„Nil przepływa przez Egipt”.
- Wykorzystaj pętlę do wyświetlenia nazw wszystkich rzek przechowywanych w słowniku.
- Wykorzystaj pętlę do wyświetlenia nazw wszystkich państw przechowywanych w słowniku.

**6.6. Ankieta.** Użyj kodu znajdującego się w programie *favorite\_languages.py*, utworzonym nieco wcześniej w tym rozdziale.

- Utwórz listę osób, które powinny wziąć udział w ankiecie dotyczącej ulubionego języka programowania. Umieść na niej pewne osoby, które już znajdują się w słowniku, oraz te, które jeszcze nie zostały zapisane w słowniku.
- Przeprowadź iterację przez listę osób, które powinny wziąć udział w ankiecie. Jeżeli dana osoba już wzięła udział w ankiecie, wyświetl komunikat z podziękowaniem za jej zaangażowanie. Natomiast jeśli dana osoba jeszcze nie udzieliła odpowiedzi w ankiecie, wyświetl komunikat z zaproszeniem do wzięcia w niej udziału.

## Zagnieżdżanie

Czasami zachodzi potrzeba przechowywania zestawu słowników na liście lub listy elementów jako wartości słownika. Mamy wówczas do czynienia z *zagnieżdżaniem*. Istnieje możliwość zagnieżdżenia zestawu słowników na liście, listy elementów wewnątrz słownika lub nawet słownika wewnątrz innego słownika. Zagnieżdżanie to funkcja o potężnych możliwościach, o czym się przekonasz, analizując następujące przykłady.

### Lista słowników

Słownik `alien_0` zawiera wiele różnych informacji o jednym obcym, ale nie ma już miejsca do przechowywania informacji o drugim obcym, nie wspominając już o ekranie pełnym obcych. W jaki sposób można zarządzać flotą obcych? Jednym z rozwiązań jest utworzenie listy obcych, na której każdy obcy będzie przedstawiony za pomocą słownika zawierającego informacje o nim. Na przykład w przedstawionym poniżej kodzie mamy listę dotyczącą trzech obcych.

*Plik aliens.py:*

---

```
alien_0 = {'color': 'zielony', 'points': 5}
alien_1 = {'color': 'żółty', 'points': 10}
alien_2 = {'color': 'czerwony', 'points': 15}
```

```
aliens = [alien_0, alien_1, alien_2] ❶
for alien in aliens:
    print(alien)
```

---

Najpierw tworzymy trzy słowniki, z których każdy przedstawia innego obcego. Polecenie w wierszu ❶ umieszcza wszystkie słowniki na liście o nazwie `aliens`. Na końcu przeprowadzamy iterację przez listę i wyświetlamy informacje o każdym obcym:

```
{'color': 'zielony', 'points': 5}
{'color': 'żółty', 'points': 10}
{'color': 'czerwony', 'points': 15}
```

---

Znacznie bardziej rzeczywisty przykład dotyczy utworzenia więcej niż tylko trzech obcych za pomocą kodu, który będzie ich automatycznie generował. Spójrz na poniższy fragment kodu, w którym wykorzystujemy funkcję `range()` do przygotowania floty 30 obcych:

```
# Utworzenie pustej listy przeznaczonej do przechowywania obcych.
aliens = []

# Utworzenie 30 zielonych obcych.
for alien_number in range(30): ❶
    new_alien = {'color': 'zielony', 'points': 5, 'speed': 'wolno'} ❷
    aliens.append(new_alien) ❸

# Wyświetlenie pierwszych pięciu obcych.
for alien in aliens[:5]: ❹
    print(alien)
print("...")

# Wyświetlenie całkowitej liczby utworzonych obcych.
print(f"Całkowita liczba obcych: {len(aliens)}")
```

---

Ten przykład rozpoczyna się od pustej listy przeznaczonej do przechowywania wszystkich obcych, którzy zostaną utworzeni. W wierszu ❶ wynikiem działania funkcji `range()` jest zbiór liczb wskazujący Pythonowi, ile razy ma być powtórzona pętla. W trakcie każdej iteracji pętli tworzymy nowego obcego (patrz wiersz ❷), a następnie dodajemy go do listy `aliens` (patrz wiersz ❸). Z kolei w wierszu ❹ używamy wycinka do wyświetlenia pierwszych pięciu obcych. Na końcu wyświetlamy wielkość listy, co potwierdza wygenerowanie pełnej floty 30 obcych:

---

```
{'color': 'zielony', 'points': 5, 'speed': 'wolno'}  
...
```

Całkowita liczba obcych: 30

---

Wprawdzie każdy wygenerowany obcy ma tę samą charakterystykę, ale każdego z nich Python uznał za oddzielnego obiektu, co pozwala nam na modyfikację poszczególnych obcych.

Jak można pracować z tego rodzaju zbiorem obcych? Wyobraź sobie, że jednym z aspektów gry jest zmiana koloru obcego i jego szybkości wraz z postępem poczynionym przez gracza. Kiedy nadchodzi czas zmiany koloru, można wykorzystać pętlę `for` i polecenie `if` do zmiany koloru obcego. Na przykład aby zmienić kolor pierwszych trzech obcych na żółty, ich szybkość na średnią, a wartość na 10 punktów, możesz skorzystać z przedstawionego poniżej kodu:

---

```
# Utworzenie pustej listy przeznaczonej do przechowywania obcych.  
aliens = []  
  
# Utworzenie 30 zielonych obcych.  
for alien_number in range (30):  
    new_alien = {'color': 'zielony', 'points': 5, 'speed': 'wolno'}  
    aliens.append(new_alien)  
  
for alien in aliens[0:3]:  
    if alien['color'] == 'zielony':  
        alien['color'] = 'żółty'  
        alien['speed'] = 'średnio'  
        alien['points'] = 10  
  
# Wyświetlenie pierwszych pięciu obcych:  
for alien in aliens[:5]:  
    print(alien)  
print("...")
```

---

Ponieważ chcemy zmodyfikować jedynie trzech pierwszych obcych, przeprowadzamy iterację przez wycinek zawierający trzy pierwsze elementy listy `aliens`. Obecnie wszyscy obcy są koloru zielonego, ale przecież nie zawsze tak będzie. Dlatego też w kodzie umieszczamy polecenie `if` dające pewność, że zmodyfikujemy jedynie zielonych obcych. Jeżeli obcy ma kolor zielony, zmieniamy go na żółty, a szybkość poruszania się obcego na średnią. Ponadto po zestrzeleniu takiego obcego gracza otrzyma 10 punktów, jak to wynika z poniższych danych wyjściowych:

---

```
{'color': 'żółty', 'points': 10, 'speed': 'średnio'}
{'color': 'żółty', 'points': 10, 'speed': 'średnio'}
{'color': 'żółty', 'points': 10, 'speed': 'średnio'}
{'color': 'zielony', 'points': 5, 'speed': 'wolno'}
{'color': 'zielony', 'points': 5, 'speed': 'wolno'}
...

```

---

Tę pętlę można jeszcze bardziej rozbudować przez dodanie bloku `elif` zmieniającego żółtego obcego w czerwonego, który porusza się szybko i jest wart 15 punktów. Poniżej przedstawiam jedynie fragment programu, w którym wprowadzamy tę zmianę:

---

```
for alien in aliens[0:3]:
    if alien['color'] == 'zielony':
        alien['color'] = 'żółty'
        alien['speed'] = 'średnio'
        alien['points'] = 10
    elif alien['color'] == 'żółty':
        alien['color'] = 'czerwony'
        alien['speed'] = 'szymbko'
        alien['points'] = 15

```

---

Bardzo często zdarza się przechowywać pewną liczbę słowników na liście, gdy każdy z tych słowników zawiera wiele rodzajów informacji dotyczących jednego obiektu. Przykładowo można utworzyć słownik dla każdego użytkownika witryny internetowej, tak jak w programie `user.py` przedstawionym wcześniej w tym rozdziale, a następnie przechowywać poszczególne słowniki na liście o nazwie `users`. Wszystkie słowniki na liście powinny mieć identyczną strukturę, aby można było przeprowadzić iterację listy i pracować z każdym obiektem słownika w taki sam sposób.

## **Lista w słowniku**

Zamiast umieszczać słownik na liście, czasami użyteczne będzie umieszczenie listy w słowniku. Zastanówmy się na przykład nad sposobem przedstawienia pizzy zamawianej przez klienta. Jeżeli do dyspozycji mamy jedynie listę, wówczas tak naprawdę możemy przechowywać tylko dodatki wybrane przez klienta. Natomiast w przypadku słownika lista dodatków będzie jednym z aspektów pizzy, o których informacje możemy przechowywać.

W poniższym fragmencie kodu dla każdej pizzy przechowujemy dwa rodzaje informacji: grubość ciasta oraz listę dodatków. Wspomniana lista dodatków to wartość przypisana kluczowi `'toppings'`. Aby użyć elementu z tej listy, należy podać nazwę słownika i klucza `'toppings'`, podobnie jak to się robi w przypadku dowolnej wartości słownika. Zamiast pojedynczej wartości otrzymujemy listę dodatków.

## Plik pizza.py:

---

```
# Przechowywanie informacji o pizzy zamawianej przez klienta.  
pizza = {  
    'crust': 'grubym',  
    'toppings': ['pieczarki', 'podwójny ser'],  
}  
  
# Podsumowanie zamówienia.  
print(f"Zamówiłeś pizzę na {pizza['crust']} cieście " ❶  
      "wraz z następującymi dodatkami:")  
  
for topping in pizza['toppings']: ❷  
    print(f"\t{topping}")
```

---

Rozpoczynamy od utworzenia słownika przeznaczonego na przechowywanie informacji o zamawianej pizzy. Jednym z kluczów słownika jest 'crust', któremu przypisaliśmy wartość w postaci ciągu tekstu 'grubym'. Kolejny klucz 'toppings' ma wartość w postaci listy przechowującej wszystkie dodatki wybrane przez klienta. W wierszu ❶ generujemy podsumowanie zamówienia, zanim rozpoczęniemy przygotowywanie pizzy. Gdy zachodzi potrzeba podziału długiego wiersza w wywołaniu `print()`, wybierz odpowiedni punkt podziału wiersza i zakończ go znakiem cytowania. Przejdź do następnego wiersza, dodaj wcięcie, dodaj otwierający znak cytowania i kontynuuj ciąg tekstowy. Python automatycznie połączy wszystkie ciągi tekstowe znalezione w nawiasie wywołania `print()`. W celu wyświetlenia dodatków tworzymy pętlę `for` (patrz wiersz ❷). Aby uzyskać dostęp do listy dodatków, używamy klucza 'toppings', a Python pobiera listę dodatków ze słownika.

Poniższe dane wyjściowe wskazują, jaka powinna być pizza, którą zamierzamy przygotować:

---

```
Zamówiłeś pizzę na grubym cieście wraz z następującymi dodatkami:  
  pieczarki  
  podwójny ser
```

---

Istnieje możliwość zagnieżdżenia listy wewnętrz słownika za każdym razem, gdy zachodzi konieczność przypisania więcej niż tylko jednej wartości pojedynczemu kluczowi w słowniku. W omawianym wcześniej przykładzie z ulubionym językiem programowania, gdybyśmy mieli możliwość przechowywania odpowiedzi respondenta na liście, każdy z nich mógłby wskazać więcej niż tylko jeden ulubiony język programowania. Podczas iteracji przez słownik program przypisałby poszczególnym osobom jako wartość listę języków, a nie tylko jeden język. Wewnątrz pętli `for` słownika używamy więc następnej pętli `for`, tym razem do przeprowadzenia iteracji listy języków podanych przez poszczególne osoby.

## Plik favorite\_languages.py:

---

```
favorite_languages = {  
    'janek': ['python', 'rust'],  
    'sara': ['c'],  
    'edward': ['rust', 'go'],  
    'paweł': ['python', 'haskell'],  
}  
  
for name, languages in favorite_languages.items(): ❶  
    print(f"\nUlubione języki programowania użytkownika {name.title()} to:")  
    for language in languages: ❷  
        print(f"\t{language.title()}")
```

---

Wartością przypisywaną poszczególnym osobom jest teraz lista. Zwróć uwagę na to, że część osób podaje tylko jeden ulubiony język programowania, podczas gdy inne kilka. W trakcie iteracji przez słownik (patrz wiersz ❶) zmiennej o nazwie `languages` używamy do przechowywania każdej wartości ze słownika, ponieważ teraz wiemy, że będzie listą. Wewnątrz głównej pętli słownika wykorzystujemy inną pętlę `for` (patrz wiersz ❷) do przeprowadzenia iteracji przez listę ulubionych języków każdej osoby. W tym momencie respondent może podać dowolną liczbę ulubionych języków programowania:

---

Ulubione języki programowania użytkownika Janek to:

Python  
Rust

Ulubione języki programowania użytkownika Sara to:

C

Ulubione języki programowania użytkownika Edward to:

Rust  
Go

Ulubione języki programowania użytkownika Paweł to:

Python  
Haskell

---

Aby jeszcze bardziej dopracować program, możemy na początku pętli `for` przeprowadzającej iterację przez słownik dodać polecenie `if` sprawdzające, czy dana osoba wskazała więcej niż tylko jeden ulubiony język programowania. Wspomniane sprawdzenie odbywa się przez analizę wartości wyniku wywołania `len(languages)`. Jeżeli osoba podała więcej niż tylko jeden ulubiony język programowania, dane wyjściowe pozostaną takie same. Natomiast w przypadku podania tylko jednego ulubionego języka, zmienimy treść komunikatu, na przykład na następujący: „Ulubiony język programowania użytkownika Sara to C”.

**UWAGA** Nie powinieneś za bardzo zagnieździć list i słowników. Jeżeli zagnieźdzasz elementy w większym stopniu, niż pokazałem we wcześniejszych przykładach, lub pracujesz z utworzonym przez innych kodem ze znacznym poziomem zagnieżdżenia, prawdopodobnie oznacza to, że istnieje prostszy sposób rozwiązania danego problemu.

## Słownik w słowniku

Można zagnieździć słownik w innym słowniku, ale w takim przypadku kod bardzo szybko staje się dość skomplikowany. Na przykład jeśli z witryny internetowej będzie korzystało wielu użytkowników o unikatowych nazwach, nazwy tych użytkowników będzie można wykorzystać w charakterze kluczy słownika. Wówczas informacje o poszczególnych użytkownikach mogą być przechowywane przy użyciu słownika jako wartości przypisanej do nazwy użytkownika. W poniższym programie przechowujemy trzy rodzaje informacji o każdym użytkowniku: imię, nazwisko i miejscowości. Dostęp do tych informacji odbywa się za pomocą iteracji przez nazwy użytkowników i powiązane z nimi słowniki z informacjami.

Plik many\_users.py:

---

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },
    'mcurie': {
        'first': 'maria',
        'last': 'skłodowska-curie',
        'location': 'paryż',
    },
}

for username, user_info in users.items(): ❶
    print(f"\nNazwa użytkownika: {username}") ❷
    full_name = f"{user_info['first']} {user_info['last']}" ❸
    location = user_info['location']

    print(f"\tImię i nazwisko: {full_name.title()}") ❹
    print(f"\tMiejscowość: {location.title()}")
```

---

Zaczynamy od zdefiniowania słownika o nazwie `users` wraz z dwoma kluczami, po jednym dla nazw użytkowników '`aeinstein`' i '`mcurie`'. Wartością powiązaną z każdym kluczem będzie słownik zawierający imię, nazwisko oraz miejscowości. W wierszu ❶ przeprowadzamy iterację przez słownik `users`. Python przechowuje każdy klucz w zmiennej `username`, natomiast powiązany z nim słownik zostaje umieszczony w zmiennej `user_info`. Kod w wierszu ❷ pętli głównej powoduje wyświetlenie nazwy użytkownika.

W wierszu ❸ rozpoczyna się uzyskiwanie dostępu do wewnętrznego słownika. Zmienna `user_info` zawierająca słownik informacji o użytkowniku ma trzy klucze: `'first'`, `'last'` i `'location'`. Poszczególne klucze wykorzystujemy do wygenerowania elegancko sformatowanego pełnego imienia i nazwiska oraz miejscowości danej osoby. Następnie wyświetlamy podsumowanie informacji o tej osobie (patrz wiersz ❹):

---

Nazwa użytkownika: aeinstein

Imię i nazwisko: Albert Einstein

Miejscowość: Princeton

Nazwa użytkownika: mcurie

Imię i nazwisko: Maria Skłodowska-Curie

Miejscowość: Paryż

---

Zwróć uwagę, że struktura słowników utworzonych dla poszczególnych użytkowników jest identyczna. Wprawdzie to nie jest wymagane przez Pythona, ale dzięki wykorzystywaniu takiej samej struktury łatwiej jest pracować z zagnieżdzonymi słownikami. Jeżeli słowniki utworzone dla poszczególnych użytkowników miałyby inne klucze, wtedy kod wewnętrz pętli `for` byłby znacznie bardziej skomplikowany.

## ZRÓB TO SAM

**6.7. Osoby.** Pracę rozpocznij od programu stworzonego w ćwiczeniu 6.1 we wcześniejszej części rozdziału. Utwórz dwa nowe słowniki przedstawiające różne osoby, a następnie wszystkie trzy słowniki umieść na liście o nazwie `people`. Przeprowadź iterację przez listę osób i wyświetl wszystkie informacje o poszczególnych osobach.

**6.8. Zwierzęta.** Utwórz kilka słowników i nadaj im nazwy zwierząt. W poszczególnych słownikach umieść informacje o zwierzętach będących ich właścicielami. Następnie te słowniki powinny znaleźć się na liście o nazwie `pets`. Teraz przeprowadź iterację przez listę i wyświetl wszystkie informacje o poszczególnych zwierzętach.

**6.9. Ulubione miejsca.** Utwórz słownik o nazwie `favorite_places`. Pomyśl o trzech imionach i użąd ich jako kluczy słownika. Każdej osobie przypisz po trzy ulubione miejsca. Aby ćwiczenie stało się jeszcze bardziej interesujące, możesz poprosić przyjaciół o podanie ulubionych miejsc. Przeprowadź iterację przez słownik i wyświetl imiona wszystkich osób oraz ich ulubione miejsca.

**6.10. Ulubione liczby.** Zmodyfikuj program utworzony w ćwiczeniu 6.2 we wcześniejszej części rozdziału. Po zmianach każda osoba może mieć więcej niż tylko jedną ulubioną liczbę. Wyświetl wszystkie osoby oraz ich ulubione liczby.

**6.11. Miasta.** Utwórz słownik o nazwie `cities`. Jako klucze podaj nazwy trzech miast. Dla każdego z nich utwórz oddzielnego słownika zawierającego informacje o danym mieście, takie jak kraj, w którym leży to miasto, przybliżona populacja oraz pewien fakt z historii tego miasta. Kluczami słownika zawierającego informacje o mieście mogą więc być '`country`', '`population`' i '`fact`'. Wyświetl nazwę każdego miasta oraz wszystkie zebrane o nim informacje.

**6.12. Rozszerzenia.** Mamy już do czynienia z przykładami skomplikowanymi na tyle, że można je rozbudowywać na wiele różnych sposobów. Wykorzystaj jeden z przykładów przedstawionych w tym rozdziale i rozbuduj go, dodając nowe klucze i wartości, zmieniając kontekst programu lub poprawiając formatowanie danych wyjściowych.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak zdefiniować słownik i pracować z umieszczonymi w nim informacjami. Zobaczyłeś, jak uzyskać dostęp do poszczególnych elementów słownika i je modyfikować, a także jak przeprowadzać iterację przez wszystkie informacje znajdujące się w słowniku. Nauczyłeś się przeprowadzać iteracje zarówno przez pary klucz-wartość słownika, jak i przez same jego klucze i wartości. Ponadto dowiedziałeś się, jak zagnieździć wiele słowników na jednej liście, wiele list w jednym słowniku oraz słowniki wewnętrz innego słownika.

W kolejnym rozdziale poznasz pętlę `while` i zobaczyisz, jak akceptować dane wejściowe pochodzące od użytkowników programu. To będzie naprawdę ekscytujący rozdział, ponieważ nauczysz się zapewniać interaktywność tworzonym programom, które wreszcie będą mogły reagować na dane wejściowe wprowadzane przez użytkowników.

# 7

## Dane wejściowe użytkownika i pętla while



WIĘKSZOŚĆ PROGRAMÓW JEST TWORZONA, ABY ROZWIAZYZWAĆ PROBLEMY UŻYTKOWNIKÓW KOŃCOWYCH. W TYM CELU ZWYKLE TRZEBIA POBRAĆ PEWNEGO RODZAJU INFORMACJE OD UŻYTKOWNIKA. PROSTYM PRZYKŁADEM tego typu sytuacji może być potrzeba ustalenia, czy dana osoba osiągnęła wiek pozwalający jej na udział w głosowaniu. Jeżeli chcesz opracować program udzielający odpowiedzi na tego rodzaju pytanie, musisz pamiętać, że program do wygenerowania właściwej odpowiedź potrzebuje informacji o wieku sprawdzanej osoby. Dlatego też użytkownikowi zostanie wyświetlony komunikat z prośbą o podanie tak zwanych *danych wejściowych* — w omawianym przykładzie jest to wiek osoby. Po otrzymaniu danych wejściowych program porównuje je z wiekiem wskazanym jako umożliwiający udział w głosowaniu i na podstawie wyniku tego porównania udziela ostatecznej odpowiedzi.

W tym rozdziale dowiesz się, jak akceptować dane wejściowe od użytkownika, aby program mógł z nimi pracować. Gdy do poprawnego działania programu potrzebne jest imię użytkownika, program może poprosić go o podanie imienia. Gdy potrzebna jest lista imion, użytkownik może zostać poproszony o podanie takiej listy. Do pobrania danych wejściowych używamy funkcji `input()`.

Zobaczysz również, jak można sprawić, aby program działał dowolnie długo, co pozwoli użytkownikom na wprowadzanie praktycznie każdej ilości informacji, które później będą mogły być przetwarzane przez program. Wykorzystamy więc pętlę `while` w Pythonie do pozostawienia programu działającego tak długo, dopóki spełnione będą określone warunki.

Mając narzędzia do pracy z danymi wejściowymi wprowadzonymi przez użytkownika oraz narzędzia do kontrolowania czasu działania programu, możemy tworzyć w pełni interaktywne aplikacje.

## Jak działa funkcja `input()`?

Funkcja `input()` wstrzymuje działanie programu i czeka na podanie pewnych informacji przez użytkownika. Wprowadzone przez niego dane wejściowe są przechowywane w zmiennej, co pozwala na wygodną pracę z nimi.

Przykładowo w przedstawionym poniżej programie prosimy użytkownika o wpisanie dowolnego tekstu, który następnie będzie z powrotem wyświetlony użytkownikowi.

Plik `parrot.py`:

---

```
message = input("Powiedz mi coś o sobie, a wyświetrzę to na ekranie: ")
print(message)
```

---

Funkcja `input()` pobiera jeden argument, jakim jest *komunikat* wyświetlany użytkownikowi, aby przekazać mu instrukcje dotyczące tego, co powinien zrobić. W omawianym przykładzie, kiedy zostanie wykonany kod w pierwszym wierszu, wyświetli się komunikat `Powiedz mi coś o sobie, a wyświetrzę to na ekranie:`. Program oczekuje na dane, które ma wprowadzić użytkownik, i kontynuuje działanie, kiedy użytkownik naciśnie klawisz `Enter`. Odpowiedź jest przechowywana w zmiennej `message`, a następnie za pomocą wywołania `print(message)` wyświetlana użytkownikowi na ekranie:

---

```
Powiedz mi coś o sobie, a wyświetrzę to na ekranie: Dzień dobry wszystkim!
Dzień dobry wszystkim!
```

---

**UWAGA** Niektóre edytory tekstu nie pozwalają na uruchamianie programów wymagających podania danych wejściowych przez użytkownika. W prawdzie możesz te edytory wykorzystać do tworzenia tego rodzaju programów, ale uruchamiać musisz je już z poziomu powłoki. Więcej informacji na temat uruchamiania programów Pythona w powłoce znajdziesz w rozdziale 1.

## Przygotowanie jasnych i zrozumiałych komunikatów

Za każdym razem, gdy używasz funkcji `input()`, powinieneś starać się przygotować jasny i zrozumiały komunikat, w którym użytkownikowi zostanie dokładnie wyjaśnione, jakiego rodzaju informacji się od niego oczekuje. Tutaj sprawdzi się każdy komunikat wskazujący użytkownikowi, jakie informacje powinien podać. Spójrz na przedstawiony poniżej przykład.

*Plik greeter.py:*

---

```
name = input("Podaj swoje imię: ")  
print(f"Witaj, {name}!")
```

---

Na końcu komunikatu umieść spację (tutaj po dwukropku), aby oddzielić komunikat od informacji wprowadzanych później przez użytkownika i jednocześnie wyraźnie wskazać miejsce, gdzie powinien je podać. Spójrz na przedstawiony poniżej przykład:

---

```
Podaj swoje imię: Eryk  
Witaj, Eryk!
```

---

Czasami trzeba przygotować komunikat znacznie dłuższy niż tylko jedna linijka tekstu. Na przykład w komunikacie dokładnie wyjaśniasz użytkownikowi, dla którego prosisz o konkretne informacje. W takim przypadku komunikat można umieścić w zmiennej, która następnie zostanie przekazana do funkcji `input()`. W ten sposób można przygotować naprawdę długi komunikat, a następnie jasne i zwięzłe wywołanie funkcji `input()`, tak jak pokazałem to poniżej.

*Plik greeter.py:*

---

```
prompt = "Jeżeli powiesz nam, kim jesteś, spersonalizujemy wyświetlany komunikat."  
prompt += "\nJak masz na imię? "  
  
name = input(prompt)  
print(f"\nWitaj, {name}!")
```

---

W powyższym fragmencie kodu pokazałem tylko jeden ze sposobów utworzenia długiego komunikatu. Wiersz pierwszy przechowuje w zmiennej `prompt` pierwszą część komunikatu. W drugim wierszu kodu za pomocą operatora `+=` pobieramy ciąg tekstowy przechowywany w tej zmiennej i na jego końcu dołączamy następny ciąg tekstowy.

Teraz komunikat zajmuje dwie linijki, a po znaku zapytania znajduje się spacja, która zapewnia lepszą czytelność tego komunikatu:

---

```
Jeżeli powiesz nam, kim jesteś, spersonalizujemy wyświetlany komunikat.  
Jak masz na imię? Eryk  
  
Witaj, Eryk!
```

---

## Użycie funkcji int() do akceptowania liczbowych danych wejściowych

Kiedy używana jest funkcja `input()`, wszystkie dane wprowadzane przez użytkownika Python interpretuje jako ciąg tekstowy. Spójrz na poniższą sesję, w której użytkownik został poproszony o podanie wieku:

```
>>> age = input("Ile masz lat? ")
Ile masz lat? 21
>>> age
'21'
```

Wprawdzie użytkownik podał liczbę 21, ale jeśli sprawdzimy wartość zmiennej `age` w Pythonie, otrzymamy wynik '`21`', czyli ciąg tekstowy przedstawiający wprowadzoną wartość liczbową. Wiemy, że Python zinterpretował dane wejściowe jako ciąg tekstowy, ponieważ liczba została ujęta w apostrofy. Jeżeli chcesz jedynie wyświetlić tego rodzaju dane wejściowe, to nie ma żadnego problemu. Jednak jeśli spróbujesz ich użyć w charakterze liczby, zostanie wygenerowany błąd, tak jak pokazałem poniżej:

```
>>> age = input("Ile masz lat? ")
Ile masz lat? 21
>>> age >= 18 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>=' not supported between instances of 'str' and 'int' ❷
```

Jeżeli wprowadzone dane wejściowe spróbujesz użyć w operacji porównania liczbowego (patrz wiersz ❶), Python wygeneruje komunikat błędu, ponieważ nie może porównać ciągu tekstuowego z liczbą całkowitą. Ciąg tekstowy '`21`' przechowywany w zmiennej `age` nie może zostać porównany z wartością liczbową `18` (patrz wiersz ❷).

Rozwiążaniem tego problemu jest użycie funkcji `int()` nakazującej Pythonowi potraktowanie danych wejściowych jako wartości liczbowych. Funkcja `int()` konwertuje liczby w postaci ciągu tekstuowego na wartość liczbową, tak jak pokazalem w poniższym fragmencie kodu:

```
>>> age = input("Ile masz lat? ")
Ile masz lat? 21
>>> age = int(age) ❶
>>> age >= 18
True
```

W omawianym przykładzie, kiedy zostają podane dane wejściowe w postaci liczby `21`, Python interpretuje je jako ciąg tekstowy, a następnie za pomocą funkcji

`int()` konwertuje na wartość liczbową (patrz wiersz ①). Teraz można bez problemów przeprowadzić dodatkowy test, czyli porównać wartość zmiennej `age` (liczba 21) z liczbą 18 i sprawdzić, czy wartość zmiennej `age` jest większa lub równa liczbie 18. Tutaj wynikiem porównania jest `True`.

Jak funkcję `int()` można wykorzystać w rzeczywistym programie? Spójrz na poniższy program ustalający, czy dana osoba jest wystarczająco wysoka, aby mogła przejechać się kolejką.

*Plik rollercoaster.py:*

---

```
height = input("Ile masz wzrostu (w centymetrach)? ")
height = int(height)

if height >= 90:
    print("\nJesteś wystarczająco wysoki na przejaźdzkę!")
else:
    print("\nBędziesz mógł się przejechać, gdy nieco urośniesz.")
```

---

Ten program porównuje wartość zmiennej `height` z liczbą 90, ponieważ wywołanie `height = int(height)` konwertuje dane wejściowe na wartość liczbową, a dopiero później przeprowadzane jest porównanie. Jeżeli podana liczba będzie większa lub równa 90, użytkownik otrzyma komunikat informujący go o wystarczającym wzroście do odbycia przejaźdzki.

---

Ile masz wzrostu (w centymetrach)? **180**

Jesteś wystarczająco wysoki na przejaźdzkę!

---

Kiedy będziesz używał liczbowych danych wejściowych do przeprowadzania obliczeń i porównań, upewnij się o wcześniejszej konwersji tych danych na wartości liczbowe.

## Operator modulo

Użytecznym narzędziem do pracy z wartościami liczbowymi jest *operator modulo* (`%`), który zwraca resztę z przeprowadzonej operacji dzielenia liczb.

---

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

---

Operator modulo nie informuje, ile razy jedna liczba zmieściła się w drugiej, a jedynie, jaka jest reszta po przeprowadzonej operacji dzielenia.

Gdy jedna liczba jest podzielna przez drugą, reszta wynosi 0, więc operator modulo zawsze zwraca 0. Ten fakt można wykorzystać do ustalenia parzystości lub nieparzystości liczby, tak jak pokazałem w poniższym programie.

*Plik even\_or\_odd.py:*

---

```
number = input("Podaj liczbę, aby dowiedzieć się, czy jest parzysta czy\nnieparzysta: ")
number = int(number)

if number % 2 == 0:
    print(f"\nLiczba {number} jest parzysta.")
else:
    print(f"\nLiczba {number} jest nieparzysta.")
```

---

Liczba parzysta jest zawsze podzielna przez dwa, więc wynikiem operatora modulo dla takiej liczby jest zero (stąd polecenie `number % 2 == 0`); w przeciwnym razie liczba jest nieparzysta:

---

```
Podaj liczbę, aby dowiedzieć się, czy jest parzysta czy nieparzysta: 42
Liczba 42 jest parzysta.
```

---

## ZRÓB TO SAM

**7.1. Wypożyczenie samochodu.** Utwórz program, który pyta użytkownika, jakiej marki samochód chciałby wypożyczyć. Następnie wyświetl komunikat wraz z podaną nazwą samochodu, na przykład: „Chwileczkę, sprawdę, czy mamy dostępny samochód Subaru”.

**7.2. Stolik w restauracji.** Utwórz program pytający klienta, na ile osób chce zarezerwować stolik. Jeżeli odpowiedzią jest liczba większa niż 8, powinieneś wyświetlić komunikat o konieczności zaczekania na stolik. W przeciwnym razie poinformuj klienta, że stolik jest gotowy.

**7.3. Wielokrotność dziesięciu.** Poproś użytkownika o podanie dowolnej liczby, a następnie sprawdź, czy jest ona wielokrotnością liczby 10.

# Wprowadzenie do pętli while

Pętla `for` pobiera zbiór elementów, a następnie jednokrotnie wykonuje blok kodu dla każdego elementu zbioru. Z kolei pętla `while` działa dopóty, dopóki zdefiniowany warunek przyjmuje wartość `True`.

## Pętla while w działaniu

Pętlę while można wykorzystać do wyświetlenia serii liczb. Przykładowo poniższy program powoduje wyświetlenie liczb od 1 do 5.

Plik counting.py:

---

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

---

W pierwszym wierszu programu rozpoczynamy odliczanie od 1 przez przyisanie zmiennej `current_number` wartości 1. Pętla `while` będzie działać dopóty, dopóki wartość zmiennej `current_number` jest mniejsza lub równa 5. Kod w bloku pętli wyświetla bieżącą wartość tej zmiennej, a następnie dodaje do niej 1: `current_number += 1`. (Operator `+=` jest tutaj skrótem dla zapisu `current_number = current_number + 1`).

Python wykonuje tę pętlę, dopóki warunek `current_number <= 5` przyjmuje wartość True. Ponieważ 1 jest mniejsze niż 5, Python wyświetla 1 i dodaje wartość 1 do zmiennej, która teraz ma już wartość 2. Jednak 2 jest nadal mniejsze niż 5, więc Python wyświetla 2 i ponownie dodaje 1 do zmiennej, która teraz ma już wartość 3, itd. Gdy wartość zmiennej `current_number` będzie większa niż 5, warunek nie zostanie spełniony i wykonywanie pętli oraz programu zostanie zakończone:

---

```
1
2
3
4
5
```

---

Używane przez Ciebie na co dzień programy na pewno zawierają pętle `while`. Przykładowo gra potrzebuje pętli `while`, aby rozgrywka mogła się toczyć, dopóki gracz ma na to ochotę i dopóki nie wyda odpowiedniego polecenia o jej zakończeniu. Programy nie byłyby zbyt użyteczne, gdyby przestawały działać przedwcześnie lub kontynuowały działanie nawet wtedy, gdy użytkownik chce je zamknąć. Dlatego też pętle `while` okazują się niezwykle przydatne.

## Umożliwienie użytkownikowi podjęcia decyzji o zakończeniu działania programu

Program `parrot.py` może działać tak długo, jak tego chce użytkownik. W tym celu logikę programu musimy umieścić wewnątrz pętli `while`. W poniższym programie zdefiniujemy tak zwaną *wartość wyjścia* — program będzie działał, dopóki użytkownik nie wprowadzi tej wartości.

## Plik parrot.py:

---

```
prompt = "\nPowiedz mi coś o sobie, a wyświetrzę to na ekranie:\n"
prompt += "\nNapisz 'koniec', aby zakończyć działanie programu.\n"

message = ""
while message != 'koniec':
    message = input(prompt)
    print(message)
```

---

Na początku programu definiujemy komunikat oferujący użytkownikowi dwie możliwości: wprowadzenie informacji lub podanie wartości wyjścia (tutaj to słowo koniec) w celu zakończenia działania programu. Następnie definiujemy zmienną `message` przeznaczoną na dane wejściowe użytkownika. Obecnie wartością tej zmiennej jest pusty ciąg tekstowy (""), więc Python i tak ma pewną wartość do sprawdzenia, gdy będzie po raz pierwszy wykonywać pętlę `while`. Kiedy zostanie uruchomiony program, w trakcie pierwszego sprawdzenia pętli `while` konieczne będzie porównanie wartości zmiennej `message` z ciągiem tekstowym 'koniec', mimo że użytkownik nie wprowadził jeszcze żadnych danych wejściowych. W przypadku braku wartości do porównania Python nie będzie mógł kontynuować działania programu. Rozwiązaniem problemu jest upewnienie się, że zmiennej `message` została przypisana wartość początkowa. Wprawdzie to jedynie pusty ciąg tekstowy, ale tak zapisana wartość ma sens dla Pythona i pozwala mu przeprowadzić porównanie, które zapewni działanie pętli `while`. Zdefiniowana pętla `while` działa tak długo, dopóki wartością zmiennej `message` nie jest 'koniec'.

W trakcie pierwszej iteracji pętli zmienna `message` jest po prostu pustym ciągiem tekstowym, więc Python wchodzi do pętli. W wierszu `message = input(message)` Python wyświetla komunikat i czeka na wprowadzenie danych wejściowych przez użytkownika. Niezależnie od tego, jaką wartość wprowadzi użytkownik, zostanie ona umieszczona w zmiennej `message` i wyświetlona na ekranie. Później Python ponownie sprawdza warunek w pętli `while`. Jeżeli nie zostało wprowadzone słowo 'koniec', komunikat zostaje wyświetlony ponownie i Python czeka na kolejne dane wejściowe. Kiedy użytkownik zdecyduje się wpisać 'koniec', Python zakończy wykonywanie pętli `while` i tym samym zakończy się działanie programu:

---

```
Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. Dzień dobry wszystkim!  
Dzień dobry wszystkim!
```

```
Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. Witaj ponownie.  
Witaj ponownie.
```

---

Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. **koniec**  
koniec

---

Ten program działa doskonale, ale niestety wyświetla słowo 'koniec' tak, jakby było ono rzeczywistym komunikatem. Rozwiązaniem jest prosty test przeprowadzany za pomocą polecenia `if`:

---

```
prompt = "\nPowiedz mi coś o sobie, a wyświetrzę to na ekranie:"  
prompt += "\nNapisz 'koniec', aby zakończyć działanie programu. "  
  
message = ""  
while message != 'koniec':  
    message = input(prompt)  
  
    if message != 'koniec':  
        print(message)
```

---

Teraz przed wyświetleniem komunikatu program przeprowadza prostą operację sprawdzenia. Komunikat jest wyświetlany tylko wtedy, gdy jego treść nie odpowiada zdefiniowanej wcześniej wartości wyjścia:

---

Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. **Dzień dobry wszystkim!**  
Dzień dobry wszystkim!

Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. **Witaj ponownie.**  
Witaj ponownie.

Powiedz mi coś o sobie, a wyświetrzę to na ekranie:  
Napisz 'koniec', aby zakończyć działanie programu. **koniec**

---

## Użycie flagi

W poprzednim przykładzie przygotowaliśmy program wykonujący określone zadania, gdy zdefiniowany warunek zostanie spełniony. Jednak co możemy zrobić w przypadku bardziej skomplikowanych programów, w których wiele różnych zdarzeń może prowadzić do zakończenia działania aplikacji?

Rozważmy grę, w której wiele różnych zdarzeń może spowodować jej zakończenie. Gra powinna zostać zakończona na przykład wtedy, kiedy gracz nie będzie już dysponował żadnym statkiem albo straci wszystkie miasta, które miał chronić, lub po prostu wtedy, kiedy upłynie czas przeznaczony na rozgrywkę. Tak więc gra ma się zakończyć po wystąpieniu jednego z wymienionych zdarzeń. Jeżeli wiele zdarzeń może prowadzić do zakończenia działania programu, sprawdzanie ich wszystkich w pojedynczej pętli `while` wydaje się niepotrzebnie skomplikowane i trudne.

W przypadku programu, który działa dopóty, dopóki określone warunki są spełnione, można zdefiniować pojedynczą zmienną wskazującą, czy cały program pozostaje aktywny. Tego rodzaju zmienna jest określana mianem *flagi* i jest rodzajem sygnału dla programu. Możemy tworzyć programy, które będą działać tak długo, jak flaga ma przypisaną wartość `True`, a zakończą swoje działanie, gdy dowolne ze zdarzeń spowoduje przypisanie flagie wartości `False`. W takim podejściu ogólne polecenie `while` musi sprawdzać tylko jeden warunek: czy flaga ma aktualnie przypisaną wartość `True`. Następnie wszystkie inne testy (sprawdzające, czy dane zdarzenie spowodowało przypisanie flagie wartości `False`) mogą być elegancko umieszczone w pozostałej części programu.

Dodamy teraz flagę do utworzonego wcześniej programu *parrot.py*. Ta flaga, tutaj o nazwie `active` (choć możesz użyć dowolnej nazwy), będzie wskazywała, czy program powinien nadal działać:

---

```
prompt = "\nPowiedz mi coś o sobie, a wyświetrzę to na ekranie:"
prompt += "\nNapisz 'koniec', aby zakończyć działanie programu. "

active = True
while active: ❶
    message = input(prompt)

    if message == 'koniec':
        active = False
    else:
        print(message)
```

---

Zmiennej `active` przypisujemy wartość `True`, więc program rozpoczyna działanie w stanie aktywnym. W ten sposób upraszczamy polecenie `while`, ponieważ nie ma potrzeby przeprowadzania w nim żadnej operacji porównania, a logika została przeniesiona do innych fragmentów programu. Dopóki wartością zmiennej `active` pozostaje `True`, pętla będzie kontynuowała działanie (patrz wiersz ❶).

W poleceniu `if` znajdującym się w bloku kodu sprawdzamy wartość zmiennej `message` po wprowadzeniu danych wejściowych przez użytkownika. Jeżeli użytkownik napisał '`koniec`', zmiennej `active` przypisujemy wartość `False`, co powoduje zakończenie działania pętli. Natomiast jeśli użytkownik wprowadził inne dane wejściowe niż słowo '`koniec`', wyświetlamy je jako treść komunikatu.

Nowa wersja programu generuje dokładnie takie same dane wyjściowe jak w poprzednim przykładzie, w którym test warunkowy znajdował się bezpośrednio w poleceniu `while`. Obecnie mamy flagę wskazującą, czy program znajduje się w stanie aktywnym, i dlatego możemy bardzo łatwo dodawać kolejne testy (na przykład w postaci poleceń `elif`) dla zdarzeń, które powinny powodować przypisanie wartości `False` zmiennej `active`. Tego rodzaju podejście okazuje się użyteczne w skomplikowanych programach, takich jak gry, w których istnieje wiele zdarzeń prowadzących do zakończenia działania aplikacji. Kiedy jedno z tych zdarzeń spowoduje przypisanie zmiennej `active` wartości `False`, główna pętla gry zakończy swoje działanie. Można wówczas wyświetlić ekran informujący o zakończeniu gry i umożliwić graczowi ponowne rozpoczęcie rozgrywki.

## Użycie polecenia break do opuszczenia pętli

Aby natychmiast opuścić pętlę `while` bez wykonywania jakiegokolwiek pozostałoego w niej kodu i niezależnie od wyniku testu warunkowego, można użyć polecenia `break`. To polecenie określa przepływ kontroli działania programu: można je wykorzystać do wskazania wykonywanych i niewykonywanych wierszy kodu po to, aby program wykonywał jedynie ten kod, który chcesz uruchomić, i robił to wtedy, kiedy tego chcesz.

Spójrz na poniższy program proszący użytkownika o podanie miejsc, które odwiedził. Działanie pętli `while` w tym programie możemy zakończyć przez wywołanie polecenia `break` natychmiast po wpisaniu przez użytkownika słowa 'koniec'.

Plik `cities.py`:

---

```
prompt = "\nPodaj nazwy miast, które chciałbyś odwiedzić:"  
prompt += "\n(Gdy zakończysz podawanie miast, napisz 'koniec').)"  
  
while True: ❶  
    city = input(prompt)  
  
    if city == 'koniec':  
        break  
    else:  
        print(f"Chciałbym odwiedzić {city.title()}!")
```

---

W wierszu ❶ pętla została zdefiniowana jako `while True`, co oznacza, że będzie działała w nieskończoność, jeśli nie zostanie wywołane polecenie `break`. Pętla nieustannie pyta użytkownika o odwiedzone przez niego miasta. Pytanie będzie się pojawiać, dopóki nie zostanie wpisane słowo 'koniec', które spowoduje wykonanie polecenia `break`, co z kolei powoduje zakończenie działania pętli:

---

```
Podaj nazwy miast, które chciałbyś odwiedzić:  
(Gdy zakończysz podawanie miast, napisz 'koniec'.) Nowy Jork  
Chciałbym odwiedzić Nowy Jork!
```

```
Podaj nazwy miast, które chciałbyś odwiedzić:  
(Gdy zakończysz podawanie miast, napisz 'koniec'.) San Francisco  
Chciałbym odwiedzić San Francisco!
```

```
Podaj nazwy miast, które odwiedziłeś:  
(Gdy zakończysz podawanie miast, napisz 'koniec'.) koniec
```

---

**UWAGA** *Polecenia `break` możesz użyć w dowolnej pętli Pythona. Na przykład polecenie `break` można wykorzystać w pętli `for` przeprowadzającej iterację przez listę lub słownik.*

## Użycie polecenia continue w pętli

Zamiast całkowicie opuścić pętlę bez wykonania pozostałego w niej kodu, za pomocą polecenia `continue` można powrócić na początek pętli, opierając się przy tym na wyniku testu warunkowego. Spójrz na poniższą pętlę odliczającą liczby od 1 do 10, ale wyświetlającą na ekranie jedynie nieparzyste liczby w podanym zakresie.

Plik `counting.py`:

---

```
current_number = 0
while current_number < 10:
    current_number += 1 ❶
    if current_number % 2 == 0:
        continue

    print(current_number)
```

---

Zaczynamy od utworzenia zmiennej `current_number` o wartości 0. Ponieważ to mniej niż 10, Python rozpoczyna wykonywanie pętli `while`. Wewnątrz pętli następuje inkrementacja wartości tej zmiennej o 1 (patrz wiersz ❶), więc aktualnie jej wartość wynosi 1. Następnie polecenie `if` sprawdza wynik działania operatora modulo zmiennej `current_number` i liczby 2. Jeżeli modulo wynosi 0 (co oznacza podzielność przez 2 liczby przechowywanej w zmiennej `current_number`), polecenie `continue` nakazuje Pythonowi zignorowanie pozostałej części pętli i powrót na jej początek. Natomiast jeśli bieżąca liczba nie jest bez reszty podzielna przez 2, pozostała część pętli jest wykonywana, a bieżąca liczba zostaje wyświetlona na ekranie:

---

```
1
3
5
7
9
```

---

## Unikanie pętli działającej w nieskończoność

Każda pętla `while` powinna mieć możliwość zakończenia działania, aby nie była uruchomiona w nieskończoność. Spójrz na poniższą pętlę wyświetlającą liczby od 1 do 5.

Plik `counting.py`:

---

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

---

Jeżeli przez przypadek pominiesz wiersz `x += 1`, jak w następnym przykładzie, wówczas utworzysz pętlę działającą w nieskończoność:

---

```
# Ta pętla działa w nieskończoność!
x = 1
while x <= 5:
    print(x)
```

---

W powyższym fragmencie kodu wartością początkową `x` jest 1, ale ta wartość nigdy nie ulega zmianie. Dlatego też test warunkowy `x <= 5` zawsze zwróci `True` i pętla `while` będzie działała w nieskończoność, wyświetlając serię 1, tak jak pokazalem poniżej:

---

```
1
1
1
1
--cięcie--
```

---

Każdemu programiście zdarza się od czasu do czasu utworzyć działającą w nieskończoność pętlę `while`, zwłaszcza gdy warunek powodujący zakończenie pętli nie jest jasno zdefiniowany. Jeżeli uruchomiony program ma pętlę działającą w nieskończoność, naciśnij klawisze `Ctrl+C`, aby zamknąć okno terminala wyświetlającego dane wyjściowe tego programu.

By uniknąć utworzenia pętli działającej w nieskończoność, dokładnie przetestuj każdą pętlę `while` i upewnij się, że kończy ona swoje działanie w oczekiwany momencie. Jeżeli chcesz zakończyć działanie programu wtedy, kiedy użytkownik wprowadzi określona wartość danych wejściowych, uruchom program i wprowadź tę wartość. Jeżeli mimo wszystko program nadal działa, dokładnie przeanalizuj sposób, w jaki program obsługuje tę wartość, która ma powodować opuszczenie pętli. Upewnij się, że przynajmniej jeden z fragmentów kodu może spowodować przypisanie wartości `False` warunkowi pętli lub doprowadzić do wykonania polecenia `break`.

**UWAGA** VS Code, podobnie jak wiele innych edytorów tekstu, ma osadzone okno danych wyjściowych. Aby zatrzymać pętlę działającą w nieskończoność, spróbuj kliknąć to okno i dopiero naciśnij klawisze `Ctrl+C`.

## ZRÓB TO SAM

**7.4. Dodatki do pizzy.** Utwórz pętlę proszącą użytkownika o podanie serii dodatków do pizzy. Działanie pętli powinno zostać zakończone, kiedy zostanie wpisane słowo 'koniec'. Gdy użytkownik będzie podawał kolejne dodatki, pętla powinna wyświetlać komunikat o dodaniu danego składnika do pizzy.

**7.5. Bilety do kina.** Cena biletu do kina jest uzależniona od wieku widza. Jeżeli widz ma poniżej 3 lat, bilet jest bezpłatny. Dla dzieci w wieku od 3 do 12 lat bilet kosztuje 10 zł. Dla osób wieku powyżej 12 lat cena biletu wynosi 15 zł. Utwórz pętlę proszącą użytkownika o podanie wieku, a następnie na podstawie otrzymanych danych wyświetl komunikat zawierający prawidłową cenę biletu.

**7.6. Trzy wyjścia.** Przygotuj trzy różne wersje ćwiczenia 7.4 lub 7.5, z których każde będzie przynajmniej raz wykonywać poniższe zadania:

- Użycie testu warunkowego w pętli while do zatrzymania jej działania.
- Użycie zmiennej active do kontroli długości działania pętli while.
- Użycie polecenia break do opuszczenia pętli, gdy użytkownik wpisze 'koniec'.

**7.7. Nieskończoność.** Utwórz pętlę działającą w nieskończoność i uruchom ją. (W celu zakończenia tej pętli naciśnij klawisze *Ctrl+C* lub zamknij okno terminala wyświetlającego dane wyjściowe programu).

## Użycie pętli while wraz z listami i słownikami

Jak dotąd pracowaliśmy tylko z jednym fragmentem informacji, które były dostarczane przez użytkownika. Program otrzymywał dane wejściowe, a następnie wyświetlał je lub odpowiednio na nie reagował. W trakcie kolejnej iteracji pętli while otrzymywaliśmy następną wartość danych wejściowych i odpowiadaliśmy na nią. Jednak w celu monitorowania wielu użytkowników i wielu fragmentów informacji konieczne jest użycie w pętli while list i słowników.

Wprawdzie pętla for jest efektywna do przeprowadzania iteracji przez listę, ale nie powinieneś modyfikować listy wewnątrz pętli for, ponieważ Python będzie miał wówczas trudności z monitorowaniem elementów na liście. Aby zmodyfikować listę, gdy wykonywana jest iteracja przez nią, należy użyć pętli while. Wykorzystanie pętli while wraz z listą i słownikiem pozwala na zbieranie, przechowywanie i organizowanie dużej ilości danych wejściowych w celu ich późniejszego analizowania.

### Przenoszenie elementów z jednej listy na drugą

Rozważ listę nowo zarejestrowanych użytkowników witryny internetowej, którzy jeszcze nie zostali zweryfikowani. Po przeprowadzeniu weryfikacji użytkownika należy go przenieść na listę potwierdzonych użytkowników — ale jak to zrobić? Jednym ze sposobów jest wykorzystanie pętli while, aby pobrać użytkowników z listy niezweryfikowanych, a następnie dodać ich do oddzielnej listy zweryfikowanych. Poniżej przedstawiłem przykład kodu stosującego tego rodzaju podejście.

## Plik confirmed\_users.py:

---

```
# Rozpoczynamy od użytkowników, którzy mają być zweryfikowani.  
# Tworzymy też pustą listę przeznaczoną do przechowywania zweryfikowanych  
# użytkowników.  
unconfirmed_users = ['alicja', 'bartek', 'katarzyna'] ❶  
confirmed_users = []  
  
# Weryfikujemy poszczególnych użytkowników, dopóki lista nie będzie pusta.  
# Każdego zweryfikowanego użytkownika przenosimy na oddzielną listę.  
while unconfirmed_users: ❷  
    current_user = unconfirmed_users.pop() ❸  
  
    print(f"Weryfikacja użytkownika: {current_user.title()}")  
    confirmed_users.append(current_user) ❹  
  
    # Wyświetlenie wszystkich zweryfikowanych użytkowników.  
print("\nZweryfikowano wymienionych poniżej użytkowników:")  
for confirmed_user in confirmed_users:  
    print(confirmed_user.title())
```

---

Program rozpoczynamy od zdefiniowania w wierszu ❶ listy niepotwierdzonych użytkowników (Alicja, Bartek i Katarzyna) oraz pustej listy przeznaczonej do przechowywania zweryfikowanych użytkowników. Pętla while w wierszu ❷ jest wykonywana dopóty, dopóki lista unconfirmed\_users zawiera jakiekolwiek elementy. Wewnątrz tej pętli metoda pop() usuwa niezweryfikowanego użytkownika z końca listy unconfirmed\_users (patrz wiersz ❸). Ponieważ w omawianym przykładzie element reprezentujący Katarzynę jest ostatnim elementem listy unconfirmed\_users, zostanie usunięty jako pierwszy, zapisany w zmiennej current\_user i dodany do listy confirmed\_users w wierszu ❹. Następnie ta sama procedura zostanie przeprowadzona dla Bartka, a później Alicji.

Potwierdzenie poszczególnych użytkowników symulujemy przez wyświetlenie komunikatu o weryfikacji, a następnie dodajemy ich do listy zweryfikowanych użytkowników. Gdy lista niezweryfikowanych użytkowników zmniejsza się, jednocześnie powiększa się lista potwierdzonych. Gdy lista niezweryfikowanych użytkowników stanie się pusta, działanie pętli się zakończy i zostanie wyświetlona lista zweryfikowanych użytkowników:

---

```
Weryfikacja użytkownika: Katarzyna  
Weryfikacja użytkownika: Bartek  
Weryfikacja użytkownika: Alicja
```

---

```
Zweryfikowano wymienionych poniżej użytkowników:  
Katarzyna  
Bartek  
Alicja
```

---

## Usuwanie z listy wszystkich egzemplarzy określonej wartości

W rozdziale 3. wykorzystaliśmy funkcję `remove()` do usunięcia określonej wartości z listy. Funkcja `remove()` się sprawdzała, ponieważ usuwane wartości pojawiały się tylko raz na liście. Co możemy zrobić w sytuacji, gdy z listy chcemy usunąć wszystkie wystąpienia danej wartości?

Zakładamy, że istnieje lista zwierząt, na której wielokrotnie pojawia się element 'kot'. Aby usunąć wszystkie wystąpienia tej wartości, możemy używać pętli `while` dopóty, dopóki wartość 'kot' znajduje się na liście, tak jak pokazałem w poniższym programie.

*Plik pets.py:*

---

```
pets = ['pies', 'kot', 'pies', 'złota rybka', 'kot', 'królik', 'kot']
print(pets)

while 'kot' in pets:
    pets.remove('kot')

print(pets)
```

---

Rozpoczynamy od listy zawierającej kilka wystąpień elementu 'kot'. Po wyświetleniu listy Python rozpoczyna wykonywanie pętli `while`, ponieważ wartość 'kot' przynajmniej raz pojawia się na liście. Python usuwa pierwsze wystąpienie wartości 'kot', powraca do wiersza z poleceniem `while` i ponownie wchodzi do pętli, jeśli lista zawiera jeszcze jakikolwiek element 'kot'. Kolejne wystąpienia wartości 'kot' będą usuwane z listy, dopóki ta wartość choć raz będzie się na liście pojawiać. Gdy na liście nie będzie już żadnej wartości 'kot', Python zakończy działanie pętli i ponownie wyświetli listę:

---

```
['pies', 'kot', 'pies', 'złota rybka', 'kot', 'królik', 'kot']
['pies', 'pies', 'złota rybka', 'królik']
```

---

## Umieszczenie w słowniku danych wejściowych wprowadzonych przez użytkownika

W trakcie każdej iteracji pętli `while` można pobrać niezbędną ilość danych wejściowych. Utworzymy teraz program, w którym podczas każdej iteracji pętli `while` użytkownik będzie podawał imię i odpowiedź na pytanie. Zebrane w ten sposób dane zostaną umieszczone w słowniku, ponieważ poszczególne odpowiedzi chcemy powiązać z użytkownikami, którzy ich udzielili.

*Plik mountain\_poll.py:*

---

```
responses = {}
# Ustawienie flagi wskazującej, czy ankiet jest aktywna.
polling_active = True
```

---

```

while polling_active:
    # Prośba o podanie imienia uczestnika ankiety oraz odpowiedzi na pytanie.
    name = input("\nJak masz na imię? ") ①
    response = input("Na który szczyt chciałbyś się wspiąć pewnego dnia? ")

    # Umieszczenie odpowiedzi w słowniku:
    responses[name] = response ②

    # Ustalenie, czy ktokolwiek jeszcze chce wziąć udział w ankiecie.
    repeat = input("Czy ktokolwiek inny chce wziąć udział w ankiecie? (tak / nie) ") ③
    if repeat == 'nie':
        polling_active = False

# Ankieta została zakończona i wyświetlamy jej wyniki.
print("\n--- Wyniki ankiety ---")
for name, response in responses.items(): ④
    print(f"{name} chciałby się wspiąć na {response}.")

```

---

Na początku programu definiujemy pusty słownik (`responses`) oraz ustawiamy flagę (`polling_active`) wskazującą, czy ankieta jest aktywna. Dopóki zmienienna `polling_active` ma wartość `True`, Python będzie wykonywał kod wewnątrz pętli `while`.

W bloku pętli prosimy użytkownika o podanie imienia oraz nazwy szczytu, na który chciałby się wspiąć (patrz wiersz ①). Podane przez użytkownika informacje są umieszczane w słowniku `responses` (patrz wiersz ②), a użytkownik ma możliwość kontynuować wypełnianie ankiety (patrz wiersz ③). Kiedy zostaje wpisane słowo '`tak`', program ponownie przystępuje do wykonywania pętli `while`. Wpisanie słowa '`nie`' powoduje przypisanie zmiennej `polling_active` wartości `False`, co z kolei prowadzi do zakończenia działania pętli `while`. Wówczas ostatni blok kodu rozpoczętyjący się w wierszu ④ wyświetla wynik przeprowadzonej ankiety.

Po uruchomieniu programu i podaniu kilku przykładowych odpowiedzi otrzymasz dane wyjściowe podobne do przedstawionych poniżej:

---

Jak masz na imię? **Eryk**

Na który szczyt chciałbyś się wspiąć pewnego dnia? **Rysy**

Czy ktokolwiek inny chce wziąć udział w ankiecie? (tak / nie) **tak**

Jak masz na imię? **Leszek**

Na który szczyt chciałbyś się wspiąć pewnego dnia? **Mount Everest**

Czy ktokolwiek inny chce wziąć udział w ankiecie? (tak / nie) **nie**

---

--- Wyniki ankiety ---

Eryk chciałby się wspiąć na Rysy.

Leszek chciałby się wspiąć na Mount Everest.

---

## ZRÓB TO SAM

**7.8. Bar.** Przygotuj listę o nazwie sandwich\_orders i umieść na niej nazwy różnych kanapek. Następnie przygotuj pustą listę o nazwie finished\_sandwiches. Przeprowadź iterację przez listę zamówionych kanapek i wyświetl informację o danym zamówieniu, na przykład: „Przygotowano kanapkę z tuńczykiem”. Kiedy kanapka zostanie zrobiona, reprezentujący ją element powinien zostać przeniesiony na listę finished\_sandwiches. Gdy lista sandwich\_orders będzie już pusta, wyświetl komunikat zawierający listę wszystkich zrobionych kanapek.

**7.9. Brak pastrami.** Wykorzystaj listę sandwich\_orders z ćwiczenia 7.8 i upewnij się, że na liście przynajmniej trzykrotnie pojawia się kanapka z pastrami. Gdzieś na początku programu umieść kod wyświetlający komunikat, że w barze skończyły się pastrami, a następnie za pomocą pętli while usuń wszystkie wystąpienia 'pastrami' z listy sandwich\_orders. Upewnij się, że żadna kanapka z pastrami nie zostanie umieszczona na liście finished\_sandwiches.

**7.10. Wymarzone wakacje.** Przygotuj program pytający użytkowników o ich wymarzone wakacje. Program powinien wyświetlać pytanie w stylu: „Jeżeli mógłbyś odwiedzić jedno dowolne miejsce na świecie, gdzie byś pojechał?”. Umieść w programie blok kodu odpowiedzialny za wyświetlenie wyników przeprowadzonej ankiety.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak za pomocą funkcji `input()` pozwolić użytkownikom na dostarczenie danych wejściowych do programu. Nauczyłeś się pracować z danymi wejściowymi wprowadzanymi zarówno w postaci tekstu, jak i liczb. Zobaczyłeś, jak używać pętli `while`, aby program działał tak długo, jak zechcesz. Poznałeś kilka sposobów kontrolowania przepływu działania pętli `while`, na przykład przez ustawienie flagi `active` oraz przez użycie poleceń `break` i `continue`. Dowiedziałeś się również, jak wykorzystać pętlę `while` do przeniesienia elementów z jednej listy na drugą oraz jak usunąć z listy wszystkie wystąpienia danej wartości. Na końcu nauczyłeś się stosować pętlę `while` razem ze słownikami.

W rozdziale 8. poznasz *funkcje*. Wspomniane funkcje pozwalają podzielić program na wiele mniejszych fragmentów, z których każdy wykonuje jedno określone zadanie. Funkcja może być wywoływana dowolną ilość razy, a poszczególne funkcje można przechowywać w oddzielnych plikach. Dzięki funkcjom możesz tworzyć znacznie efektywniejszy kod, który będzie łatwiejszy do debugowania i późniejszej obsługi, a także będzie nadawał się do wykorzystania w wielu różnych programach.

# 8

## Funkcje



W TYM ROZDZIALE DOWIESZ SIĘ, JAK TWORZYĆ FUNKCJE, CZYLI NAZWANE BLOKI KODU PRZEZNACZONE DO WYKONYWANIA JEDNEGO OKREŚLONEGO ZADANIA. KIEDY CHCESZ URUCHOMIĆ KONKRETNĄ OPERACJĘ zdefiniowaną w funkcji, wówczas *wywołujesz* nazwę odpowiedniej funkcji.

Jeżeli to samo zadanie ma być wykonywane wielokrotnie w programie, nie musisz ponownie wpisywać kodu, który dotyczy tego zadania. Wystarczy, że po prostu wywołasz funkcję odpowiedzialną za przeprowadzenie tej operacji, a Python natkaże wykonanie kodu zdefiniowanego w tej funkcji. Przekonasz się, że dzięki użyciu funkcji programy stają łatwiejsze do tworzenia, odczytywania, testowania i poprawiania.

Zobaczysz również w tym rozdziale, jak można przekazywać informacje do funkcji. Poznasz sposoby tworzenia tych funkcji, których podstawowym zadaniem jest wyświetlanie informacji, oraz innych funkcji, które odpowiadają za przetwarzanie danych i zwracanie wartości bądź zbioru wartości. Na końcu zajmiemy się tematem przechowywania funkcji w oddzielnich plikach (nazwanych *modułami*), które pomagają organizować główne pliki programu.

### Definiowanie funkcji

Poniżej przedstawiłem prostą funkcję o nazwie `greet_user()`, której zadaniem jest wyświetlenie powitania.

```
def greet_user():
    """Wyświetla proste powitanie."""
    print("Witaj!")

greet_user()
```

---

Ten przykład pokazuje najprostszą strukturę funkcji. Kod w pierwszym wierszu używa słowa kluczowego `def` do poinformowania Pythona, że definiujemy funkcję. To jest tak zwana *definicja funkcji*, która podaje Pythonowi nazwę funkcji i ewentualnie rodzaj informacji niezbędnych funkcji do wykonania jej zadania. Wspomniane informacje są umieszczane w nawiasie po nazwie funkcji. W omawianym przykładzie nazwa funkcji to `greet_user()`, a pusty nawias wskazuje, że do działania funkcja nie potrzebuje żadnych dodatkowych informacji — mimo tego podanie pustego nawiasu jest wymagane. Na końcu definicji znajduje się obowiązkowy dwukropiek.

Wszystkie wcięte wiersze kodu znajdujące się po definicji funkcji, tutaj po `def greet_user():`, tworzą *treść funkcji*. Komentarz widoczny w drugim wierszu to tak zwany *docstring*, jego zadaniem jest opis działania funkcji. Tego rodzaju komentarze są ujęte w potrójny cudzysłów, który jest wyszukiwany przez Pythona podczas generowania dokumentacji dla funkcji zdefiniowanych w Twoich programach.

Kolejny wiersz zawiera polecenie `print("Witaj!")`, które w tej funkcji jest jedynym rzeczywistym wierszem kodu. Dlatego też zadanie funkcji `greet_user()` polega na wyświetleniu powitania w postaci komunikatu `Witaj!`.

Kiedy chcesz użyć tej funkcji, wystarczy, że ją po prostu wywołasz. Wspomniane *wywołanie funkcji* nakazuje Pythonowi wykonanie kodu funkcji. Aby *wywołać* funkcję, należy po prostu podać jej nazwę wraz z nawiasem zawierającym wszystkie niezbędne informacje. Ponieważ w omawianym przykładzie funkcja nie wymaga podania jakichkolwiek informacji dodatkowych, wywołanie przybiera prostą postać `greet_user()`. Zgodnie z oczekiwaniami wygenerowane będą następujące dane wyjściowe:

---

Witaj!

---

## Przekazywanie informacji do funkcji

Kiedy wprowadzi się drobną modyfikację, funkcja `greet_user()` może nie tylko wyświetlić słowo `Witaj!`, lecz także imię użytkownika. W tym celu w nawiasie definicji funkcji (`def greet_user():`) należy wpisać nazwę zmiennej, np. `username`. Dodanie wymienionej zmiennej oznacza, że funkcja będzie akceptowała dowolną wprowadzoną wartość `username`. Po tej zmianie funkcja oczekuje podania wartości dla `username` za każdym razem, kiedy zostanie wywołana. Dlatego też podczas wywoływania funkcji `greet_user()` można do niej przekazać w nawiasie imię użytkownika, na przykład '`janek`':

```
def greet_user(username):
    """Wyświetla proste powitanie."""
    print(f" Witaj, {username.title()}!")

greet_user('janek')
```

Za pomocą polecenia `greet_user('janek')` wywołujemy funkcję `greet_user()` i przekazujemy jej informacje niezbędne do wykonania polecenia `print()`. Funkcja akceptuje przekazane imię, a następnie wyświetla powitanie zawierające to imię:

```
 Witaj, Janek!
```

Podobnie za pomocą polecenia `greet_user('sara')` wywołamy funkcję `greet_user()`, przekażemy jej argument '`sara`' i wyświetlimy powitanie `Witaj, Sara!`. Wywołania `greet_user()` możesz użyć dowolną ilość razy, w każdym przypadku otrzymasz przewidywalne dane wyjściowe.

## Argumenty i parametry

Przedstawioną powyżej funkcję `greet_user()` zdefiniowaliśmy w taki sposób, że oczekuje ona podania wartości dla zmiennej `username`. Kiedy wywołamy tę funkcję i przekażemy jej wymagane informacje (tutaj imię witanej osoby), zostanie wyświetlone spersonalizowane powitanie danego użytkownika.

Zmienna `username` w definicji funkcji `greet_user()` jest przykładem *parametru*, czyli pewnego fragmentu informacji, który jest wymagany przez funkcję, aby mogła ona wykonać przypisane jej zadanie. Z kolei wartość '`janek`' w wywołaniu `greet_user('janek')` jest przykładem *argumentu*. Wspomniany argument to fragment informacji przekazywany z wywołania funkcji do treści funkcji. Kiedy funkcja jest wywoływaną, wartość, z którą ma ona pracować, jest umieszczana w nawiasie. W omawianym przykładzie funkcji `greet_user()` został przekazany argument '`janek`', który następnie został umieszczony w parametrze `username`.

**UWAGA** Terminy *argument* i *parametr* są bardzo często używane wymiennie. Nie bądź więc zaskoczony, jeśli spotkasz się z użyciem słowa *argument* w odniesieniu do zmiennej w definicji funkcji lub słowa *parametr* w odniesieniu do zmiennej w wywołaniu funkcji.

### ZRÓB TO SAM

**8.1. Komunikat.** Utwórz funkcję o nazwie `display_messages()` wyświetlającą jedno zdanie informujące o tym, czego się uczysz w tym rozdziale. Wywołaj przygotowaną funkcję i upewnij się, że komunikat został prawidłowo wyświetlony.

**8.2. Ulubiona książka.** Utwórz funkcję o nazwie `favorite_book()`, która akceptuje jeden parametr `title`. Ta funkcja powinna wyświetlać komunikat w stylu: „Jedną z moich ulubionych książek jest *Alicja w krainie czarów*”. Wywołaj tę funkcję i upewnij się, że podałeś tytuł książki jako argument.

## Przekazywanie argumentów

Ponieważ definicja funkcji może zawierać wiele parametrów, wywołanie funkcji może wymagać wielu argumentów. Przekazywanie argumentów do funkcji może odbywać się na różne sposoby. Jedną z możliwości jest zastosowanie *argumentów pozycyjnych*, które muszą być ulożone w takiej samej kolejności jak zapisane parametry. Druga możliwość to *argumenty w postaci słów kluczowych*, w tym przypadku każdy argument składa się z nazwy zmiennej i wartości. Ostatnia możliwość to listy oraz słowniki wartości. Wszystkie wymienione możliwości zostaną teraz po kolei omówione.

### Argumenty pozycyjne

Podeczas wywoływania funkcji Python musi dopasować każdy argument w wywołaniu do parametru znajdującego się w definicji funkcji. Najprostsze rozwiązań tego problemu wykorzystuje kolejność dostarczonych argumentów. Wartości dopasowywane w ten sposób są nazywane *argumentami pozycyjnymi*.

Aby zobaczyć, jak to działa, rozważ funkcję wyświetlającą informacje o zwierzęciu. Omawiana tutaj funkcja podaje rodzaj zwierzęcia oraz jego imię, tak jak pokazałem w poniższym fragmencie kodu.

Plik `pets.py`:

```
def describe_pet(animal_type, pet_name): ❶
    """Wyświetla informacje o zwierzęciu."""
    print(f"\nMoje zwierzę to {animal_type}.")
    print(f"Moje {animal_type} ma na imię {pet_name.title()}.")

describe_pet('chomik', 'harry') ❷
```

Definicja omawianej funkcji pokazuje, że wymaga ona informacji o rodzaju zwierzęcia oraz jego imieniu (patrz wiersz ❶). Podeczas wywoływania `describe_pet()` konieczne jest podanie rodzaju zwierzęcia i jego imienia, dokładnie w tej kolejności. Przykładowo w wywołaniu funkcji argument '`chomik`' jest przekazywany w parametrze `animal_type`, natomiast argument '`harry`' w parametrze `pet_name` (patrz wiersz ❷). W treści funkcji te dwa parametry są używane w celu wyświetlenia informacji o danym zwierzęciu.

Wygenerowane dane wyjściowe dotyczą chomika o imieniu Harry:

---

```
Moje zwierzę to chomik.  
Mój chomik ma na imię Harry.
```

---

## Wiele wywołań funkcji

Funkcję można wywołać dowolną ilość razy. Opisanie drugiego, innego zwierzęcia wymaga po prostu kolejnego wywołania funkcji `describe_pet()`.

---

```
def describe_pet(animal_type, pet_name):  
    """Wyświetla informacje o zwierzęciu."""  
    print(f"\nMoje zwierzę to {animal_type}.")  
    print(f"Mój {animal_type} ma na imię {pet_name.title()}.")  
  
describe_pet('chomik', 'harry')  
describe_pet('pies', 'willie')
```

---

W drugim wywołaniu funkcji `describe_pet()` przekazujemy argumenty '`pies`' i '`willie`'. Tak jak w przypadku poprzedniego zestawu argumentów Python dopasowuje argument '`pies`' do parametru `animal_type`, natomiast argument '`willie`' do parametru `pet_name`. Podobnie jak wcześniej funkcja wykonuje swoje zadanie, ale tym razem wyświetla informacje dotyczące psa o imieniu Willie. W tym momencie mamy chomika Harry'ego i psa Williego:

---

```
Moje zwierzę to chomik.  
Mój chomik ma na imię Harry.
```

---

```
Moje zwierzę to pies.  
Mój pies ma na imię Willie.
```

---

Wielokrotne wywoływanie tej samej funkcji to bardzo efektywny sposób pracy. Kod przeznaczony do opisania zwierzęcia jest tworzony tylko jednokrotnie w funkcji. Następnie gdy zajdzie konieczność wyświetlenia informacji o jakimkolwiek zwierzęciu, wystarczy wywołać tę funkcję i przekazać jej odpowiednie informacje. Nawet jeśli kod odpowiedzialny za wyświetlenie informacji o zwierzęciu zostanie zapisany w 10 wierszach, zwierzę nadal będzie można opisać za pomocą pojedynczego wiersza zawierającego jedynie wywołanie tej funkcji.

## W przypadku argumentów pozycyjnych kolejność ma znaczenie

Otrzymasz nieoczekiwane wyniki, jeśli pomieszasz kolejność argumentów podczas wywoływania funkcji, w której zastosowano argumenty pozycyjne:

---

```
def describe_pet(animal_type, pet_name):
    """Wyswietla informacje o zwierzęciu."""
    print(f"\nMoje zwierzę to {animal_type}.")
    print(f"My {animal_type} ma na imię {pet_name.title()}.")

describe_pet('harry', 'chomik')
```

---

W pokazanym powyżej wywołaniu funkcji najpierw podaliśmy imię zwierzęcia, a dopiero później jego rodzaj. Ponieważ argument 'harry' został podany jako pierwszy, zostanie umieszczony w parametrze `animal_type`. Natomiast argument 'chomik' trafi do parametru `pet_name`. W ten sposób otrzymaliśmy zwierzę „harry” o imieniu „Chomik”:

---

```
Moje zwierzę to harry.
Mój harry ma na imię Chomik.
```

---

Jeżeli otrzymasz tego rodzaju nieoczekiwane wyniki, sprawdź, czy kolejność argumentów w wywołaniu funkcji odpowiada kolejnością parametrów znajdujących się w definicji tej funkcji.

## Argumenty w postaci słów kluczowych

*Argument w postaci słowa kluczowego* to para nazwa-wartość przekazywana do funkcji. Nazwę i jej wartość możesz zdefiniować bezpośrednio w argumencie, więc podczas przekazywania tego rodzaju argumentu do funkcji nie powinno być zamieszania (to znaczy nie otrzymasz komunikatu dotyczącego zwierzęcia harry o imieniu Chomik). Argumenty w postaci słów kluczowych zwalniają programistę z obowiązku przejmowania się kolejnością argumentów w wywołaniu funkcji oraz wyraźnie wskazują rolę poszczególnych wartości w wywołaniu funkcji.

Zmodyfikujemy teraz program `pets.py` tak, aby w wywołaniu `describe_pet()` były stosowane argumenty w postaci słów kluczowych:

---

```
def describe_pet(animal_type, pet_name):
    """Wyswietla informacje o zwierzęciu."""
    print(f"\nMoje zwierzę to {animal_type}.")
    print(f"My {animal_type} ma na imię {pet_name.title()}.")

describe_pet(animal_type='chomik', pet_name='harry')
```

---

Funkcja `describe_pet()` nie uległa zmianie. Jednak w trakcie jej wywoływania wyraźnie wskazujemy Pythonowi, do którego parametru powinien być dopasowany dany argument. Kiedy Python odczytuje wywołanie funkcji, zgodnie z przekazanymi wskazówkami umieszcza argument 'chomik' w parametrze `animal_type`,

natomiaszt argument 'harry' w parametrze `pet_name`. Wygenerowane dane wyjściowe pokazują, że opisywanym zwierzęciem jest chomik o imieniu Harry.

Kolejność argumentów w postaci słów kluczowych nie ma znaczenia, ponieważ Python doskonale wie, jak połączyć ze sobą argumenty i parametry. Dlatego też dwa przedstawione poniżej wywołania funkcji spowodują wygenerowanie tych samych danych wyjściowych:

---

```
describe_pet(animal_type='chomik', pet_name='harry')
describe_pet(pet_name='harry', animal_type='chomik')
```

---

**UWAGA** *Podczas stosowania argumentów w postaci słów kluczowych upewnij się, że używasz dokładnie tych samych nazw parametrów, które znajdują się w definicji funkcji.*

## Wartości domyślne

Podczas tworzenia funkcji można dla każdego parametru zdefiniować tak zwaną *wartość domyślną*. Jeżeli w wywoaniu funkcji zostanie podany dla parametru jakiś argument, Python użyje wartości tego argumentu. Natomiast w przeciwnym razie zastosuje wartość domyślną parametru. Dlatego też jeśli zdefiniujesz wartość domyślną dla parametru, będziesz mógł pominąć odpowiadający mu argument, który zwykle powinien być podawany w wywoaniu funkcji. Wykorzystanie wartości domyślnych może upraszczać wywoływanie funkcji i jednocześnie wyjaśniać sposób, w jaki zazwyczaj są używane dane funkcje.

Jeśli zauważysz na przykład, że większość wywołań funkcji `describe_pet()` jest używanych w celu opisywania psów, to możesz zdefiniować wartość domyślną 'pies' dla parametru `animal_type`. W ten sposób podczas wywoływania funkcji `describe_pet()` w celu wyświetlenia informacji o psie można pominąć argument 'pies':

---

```
def describe_pet(pet_name, animal_type='pies'):
    """Wyświetla informacje o zwierzęciu."""
    print(f"\nMoje zwierzę to {animal_type}.")
    print(f"Moje {animal_type} ma na imię {pet_name.title()}.")
```

---

```
describe_pet(pet_name='willie')
```

---

W powyższym fragmencie kodu zmodyfikowaliśmy definicję funkcji `describe_pet()`, która teraz zawiera wartość domyślną 'pies' dla parametru `animal_type`. Dlatego też wywołanie funkcji bez argumentu dla `animal_type` spowoduje, że Python użyje wartości 'pies' dla tego parametru:

---

```
Moje zwierzę to pies.
Moje pies ma na imię Willie.
```

---

Zwróć uwagę na konieczność zmiany kolejności parametrów w definicji funkcji. Ponieważ wartość domyślna sprawia, że nie ma potrzeby podawania rodzaju zwierzęcia, jedynym argumentem pozostałym w wywołaniu funkcji jest imię zwierzęcia. Python wciąż interpretuje ten argument jako argument pozycyjny i dlatego wywołanie funkcji z podaniem jedynie imienia zwierzęcia spowoduje dopasowanie tego argumentu do pierwszego parametru wymienionego na liście definicji funkcji. To jest powód, dla którego pierwszym parametrem musi być `pet_name`.

Najprostszym przykładem użycia przygotowanej powyżej funkcji jest teraz podanie w wywołaniu funkcji jedynie imienia psa, tak jak pokazałem poniżej:

---

```
describe_pet('willie')
```

---

To wywołanie spowoduje wygenerowanie takich samych danych wyjściowych jak w poprzednim przykładzie. Jedynym podanym argumentem jest '`'willie'`', który zostanie dopasowany do pierwszego parametru w definicji funkcji, czyli do parametru `pet_name`. Ponieważ dla parametru `animal_type` nie został podany żaden argument, Python używa wartości domyślnej, czyli '`'pies'`'.

Aby wyświetlić informacje o innym zwierzęciu niż pies, konieczne będzie wywołanie funkcji w pokazany poniżej sposób:

---

```
describe_pet(pet_name='harry', animal_type='chomik')
```

---

Ponieważ wyraźnie podano argument dla parametru `animal_type`, Python zignoruje wartość domyślną parametru.

**UWAGA** Jeżeli używasz wartości domyślnych, każdy parametr zawierający wartość domyślną musi być w definicji funkcji wymieniony po tych wszystkich parametrach, dla których nie zdefiniowano wartości domyślnych. To pozwala Pythonowi na dalszą prawidłową interpretację argumentów pozycyjnych.

## Odpowiedniki wywołań funkcji

Ponieważ argumenty pozycyjne, argumenty w postaci słów kluczowych oraz wartości domyślne mogą być stosowane łącznie, bardzo często istnieje wiele sposobów wywołania tej samej funkcji. Spójrz na przedstawioną poniżej definicję funkcji `describe_pets()` z podaną jedną wartością domyślną:

---

```
def describe_pet(pet_name, animal_type='pies'):
```

---

W przypadku powyższej definicji zawsze trzeba podać argument dla parametru `pet_name`, a sama wartość może zostać dostarczona za pomocą argumentu pozycyjnego lub w postaci słowa kluczowego. Jeżeli wywołanie ma wyświetlać informacje o zwierzęciu innym niż pies, trzeba podać także argument dla parametru

`animal_type`. Ten argument również może zostać podany z użyciem formatu pozycyjnego lub słowa kluczowego.

Wszystkie wymienione poniżej wywołania będą się sprawdzały dla tej funkcji:

---

```
# Pies o imieniu Willie.  
describe_pet('willie')  
describe_pet(pet_name='willie')  
  
# Chomik o imieniu Harry.  
describe_pet('harry', 'chomik')  
describe_pet(pet_name='harry', animal_type='chomik')  
describe_pet(animal_type='chomik', pet_name='harry')
```

---

Każde z powyższych wywołań funkcji spowoduje wygenerowanie takich samych danych wyjściowych jak w poprzednich przykładach.

Zupełnie nie ma znaczenia, który wybierzesz styl wywołań funkcji. Jeżeli funkcja generuje dane wyjściowe zgodne z oczekiwaniami, stosuj ten styl wywołań, który uznajesz za najłatwiejszy do zrozumienia.

## Unikanie błędów związanych z argumentami

Kiedy zaczynasz używać funkcji, nie bądź zaskoczony pojawiającymi się błędami dotyczącymi niedopasowanych argumentów. Z niedopasowaniem argumentów mamy do czynienia wtedy, gdy podajemy mniej lub więcej argumentów, niż jest to wymagane do prawidłowego działania funkcji. Zobaczmy, co się stanie, jeżeli spróbujemy wywołać funkcję `describe_pet()` bez żadnych argumentów:

---

```
def describe_pet(animal_type, pet_name):  
    """Wyświetla informacje o zwierzęciu."""  
    print(f"\nMoje zwierzę to {animal_type}.")  
    print(f"Moje {animal_type} ma na imię {pet_name.title()}.")  
  
describe_pet()
```

---

Python ustali, że w wywołaniu funkcji zabrakło pewnych danych. Informacje wyświetcone w wygenerowanym stosie wywołań wyraźnie to potwierdzają:

---

```
Traceback (most recent call last):  
  File "pets.py", line 6, in <module> ❶  
    describe_pet() ❷  
    ~~~~~~  
TypeError: describe_pet() missing 2 required positional arguments:  
  'animal_type' and 'pet_name' ❸
```

---

W wierszu ❶ stosu wywołań mamy informację o źródle problemu. Dzięki temu wiemy, że jest coś nie tak z użytym wywołaniem funkcji. Wiersz ❷ zawiera problematyczne wywołanie funkcji w takiej postaci, w jakiej zostało ono użyte. Następnie w wierszu ❸ stos wywołań informuje o braku dwóch argumentów, których nazwy

zostały tutaj wymienione. Jeżeli funkcja znajdowałaby się w oddzielnym pliku, prawdopodobnie można by było ponownie wywołać tę funkcję, tym razem poprawnie, bez konieczności przechodzenia do wspomnianego pliku i odczytywania kodu funkcji.

Python okazuje się pomocny, ponieważ odczytuje kod funkcji i podaje nazwy argumentów, które należy podać w wywołaniu funkcji. To jest następny argument za tym, aby nadawać zmiennym oraz funkcjom jasne i zrozumiałe nazwy. Jeżeli będziesz się trzymać takiego podejścia, generowane przez Pythona komunikaty błędów okażą się znacznie bardziej użyteczne zarówno dla Ciebie, jak i innych osób odczytujących przygotowany przez Ciebie kod źródłowy.

Jeżeli w wywołaniu omawianej funkcji podasz zbyt wiele argumentów, otrzymasz podobny do powyższego stos wywołań, który pomoże Ci w prawidłowym dopasowaniu wywołania funkcji do jej definicji.

## ZRÓB TO SAM

**8.3. T-shirt.** Utwórz funkcję o nazwie `make_shirt()`, akceptującą wielkość koszulki oraz tekst, który ma zostać na niej nadrukowany. Funkcja powinna wyświetlić zdanie zawierające informacje dotyczące zamówionej koszulki: jej rozmiar i tekst do wydrukowania na niej.

W trakcie pierwszego wywołania funkcji do przygotowania koszulki zastosuj argumenty pozycyjne. Natomiast w trakcie drugiego wywołania użyj argumentów w postaci słów kluczowych.

**8.4. Duże koszulki.** Zmodyfikuj funkcję `make_shirt()` tak, aby domyślnie były przygotowywane duże koszulki z nadrukowanym tekstem „Uwielbiam Pythona”. Utwórz koszulki w rozmiarze dużym i średnim (obie z domyślnym tekstem) oraz koszulkę w dowolnym rozmiarze i z innym tekstem nadrukowanym na niej.

**8.5. Miasta.** Utwórz funkcję o nazwie `describe_city()`, akceptującą nazwę miasta i kraju. Ta funkcja powinna wyświetlać proste zdanie, takie jak „Warszawa leży w Polsce”. Parametrowi przechowującemu nazwę państwa przypisz wartość domyślną. Przygotowaną funkcję wywołaj dla trzech różnych miast, z których najmniej jedno nie powinno być położone w domyślnie zdefiniowanym kraju.

# Wartość zwrotna

Funkcja nie zawsze musi bezpośrednio wyświetlać wygenerowane dane. Zamiast tego może przetwarzać pewne dane i zwracać wartość bądź zestaw wartości. Tego rodzaju wartość zwracana przez funkcję jest określana mianem *wartości zwrotnej*. Polecenie `return` pobiera wartość z wewnętrz funkcji i przekazuje ją do wiersza kodu, w którym nastąpiło wywołanie tej funkcji. Dzięki wartości zwrotnej można większość zadań wykonywanych przez program przenieść do funkcji, co niezwykle upraszcza część główną aplikacji.

## Zwrot prostej wartości

Spójrz na funkcję, która pobiera imię i nazwisko, natomiast jej wartością zwrotną jest elegancko sformatowane pełne imię i nazwisko.

Plik `formatted_name.py`:

```
def get_formatted_name(first_name, last_name):
    """Zwraca elegancko sformatowane pełne imię i nazwisko."""
    full_name = f"{first_name} {last_name}" ❶
    return full_name.title() ❷

musician = get_formatted_name('jimi', 'hendrix') ❸
print(musician)
```

Definicja funkcji `get_formatted_name()` pobiera jako parametry imię i nazwisko danej osoby. Kod funkcji łączy je ze sobą, umieszczając spację między nimi, a wynik przekazuje do zmiennej `full_name`, jak możesz zobaczyć w wierszu ❶. Następnie w wierszu ❷ wartość zmiennej `full_name` jest konwertowana za pomocą funkcji `title()`, która sprawia, że pierwsza litera każdego słowa zostaje zmieniona na wielką. Tak zmodyfikowany ciąg tekstowy zostaje zwrócony.

Podczas wywoływanego funkcji zwracającej wartość konieczne jest dostarczenie zmiennej przeznaczonej na przechowywanie tej wartości. W omawianym przykładzie wartość zwrotna zostaje umieszczona w zmiennej `musician` (patrz wiersz ❸). Dane wyjściowe pokazują elegancko sformatowane pełne imię i nazwisko wygenerowane na podstawie danych przekazanych do funkcji:

---

Jimi Hendrix

---

Może się wydawać, że konieczne było wykonanie niemałej ilości pracy, aby otrzymać elegancko sformatowane imię i nazwisko, skoro ten sam efekt można uzyskać za pomocą poniższego wywołania:

---

```
print("Jimi Hendrix")
```

---

Rozważ jednak sytuację, gdy pracujesz nad ogromnym programem wymagającym oddzielnego przechowywania imienia i nazwiska. Wówczas funkcja taka jak `get_formatted_name()` okazuje się niezwykle użyteczna. Imię i nazwisko nadal możesz przechowywać oddzielnie, a kiedy zajdzie konieczność wyświetlenia elegancko sformatowanego pełnego imienia i nazwiska, wtedy wywołasz wymienioną funkcję.

## Definiowanie argumentu jako opcjonalnego

Czasami sensowne jest zdefiniowanie argumentu jako opcjonalnego, aby osoby używające danej funkcji mogły zdecydować się na dostarczenie informacji dodatkowych tylko wtedy, gdy będą tego chcięły. Dzięki użyciu wartości domyślnych argument może stać się opcjonalny.

Przykładowo chcemy rozbudować funkcję `get_formatted_name()`, aby zapewnić jej możliwość obsługi drugiego imienia bądź inicjału. Pierwsza próba modyfikacji funkcji może przedstawić się następująco:

---

```
def get_formatted_name(first_name, middle_name, last_name):
    """Zwraca elegancko sformatowane pełne imię i nazwisko."""
    full_name = f'{first_name} {middle_name} {last_name}'
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

---

Ta funkcja sprawdza się doskonale po podaniu obu imion i nazwiska. Funkcja pobiera wszystkie trzy elementy pełnego imienia i nazwiska, a następnie na ich podstawie tworzy ciąg tekstowy. Między poszczególnymi elementami znajdują się umieszczone spacje, a pierwsza litera każdego elementu zostaje zmieniona na wielką:

---

John Lee Hooker

---

Jednak drugie imię lub inicjał nie zawsze są stosowane, więc powyższa funkcja nie sprawdzi się w przypadku, kiedy zostanie wywołana jedynie z imieniem i nazwiskiem. Aby drugie imię było opcjonalne, argumentowi `middle_name` trzeba przypisać wartość domyślną w postaci pustego ciągu tekstowego i zignorować ją, jeśli użytkownik nie dostarczy wartości dla wymienionego argumentu. Dlatego też by zapewnić prawidłowe działanie funkcji `get_formatted_name()` bez drugiego imienia, argumentowi `middle_name` przypisujemy wartość domyślną w postaci pustego ciągu tekstowego i przenosimy go na koniec listy parametrów:

---

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Zwraca elegancko sformatowane pełne imię i nazwisko."""
    if middle_name: ❶
        full_name = f'{first_name} {middle_name} {last_name}'
    else: ❷
        full_name = f'{first_name} {last_name}'
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

musician = get_formatted_name('john', 'hooker', 'lee') ❸
print(musician)
```

---

W powyższym przykładzie pełne imię i nazwisko jest tworzone na podstawie trzech elementów. Ponieważ zawsze zostaną podane imię i nazwisko, te parametry są wymienione jako pierwsze w definicji funkcji. Drugie imię jest opcjonalne, więc odpowiadający mu parametr został zapisany jako ostatni w definicji funkcji i ma wartość domyślną w postaci pustego ciągu tekstowego.

W treści funkcji sprawdzamy, czy podane zostało drugie imię. Niepusty ciąg tekstowy jest przez Pythona interpretowany jako wartość True i dlatego polecenie `if middle_name` przyjmie wartość True, jeśli w wywołaniu funkcji znajdzie się argument dla drugiego imienia (patrz wiersz ①). W przypadku podania drugiego imienia wszystkie trzy komponenty zostaną wykorzystane do przygotowania pełnego imienia i nazwiska. Pierwsza litera każdego komponentu będzie za pomocą funkcji `title()` zmieniona na wielką. Następnie wygenerowana wartość zwrotna zostanie przekazana do wiersza, w którym nastąpiło wywołanie funkcji. Wynik zostanie umieszczony w zmiennej `musician`, a później wyświetlony na ekranie. Jeżeli drugie imię nie zostanie podane, pusty ciąg tekstowy spowoduje, że polecenie `if` przyjmie wartość False i dlatego zostanie wykonany blok `else` (patrz wiersz ②). Pełne imię i nazwisko zostanie utworzone jedynie na podstawie podanego imienia i nazwiska, a po sformatowaniu zostanie przekazane do wiersza kodu wywołującego tę funkcję. Podobnie jak wcześniej wynik zostanie umieszczony w zmiennej `musician`, a następnie wyświetlony na ekranie.

Wywołanie omawianej funkcji jedynie wraz z imieniem i nazwiskiem jest proste. Natomiast jeśli podane ma być drugie imię, trzeba się upewnić, że zostało dostarczone na końcu, aby tym samym Python mógł prawidłowo dopasować argumenty pozycyjne ③.

Ta zmodyfikowana wersja wcześniejszej funkcji sprawdza się zarówno w przypadku osób mających tylko jedno imię i nazwisko, jak i w przypadku osób posiadających drugie imię:

---

```
Jimi Hendrix  
John Lee Hooker
```

---

Dzięki wartościom opcjonalnym funkcja obsługuje szerokie spektrum sytuacji, a jednocześnie jej kod zyskuje najprostszą możliwą postać.

## Zwrot słownika

Wartość zwrotna funkcji może być dowolnego rodzaju, w tym również może się zaliczać do bardziej skomplikowanych struktur danych, takich jak na przykład lista lub słownik. W przedstawionym poniżej programie funkcja pobiera imię i nazwisko, a zwraca słownik zawierający informacje o danej osobie.

*Plik person.py:*

---

```
def build_person(first_name, last_name):  
    """Zwraca słownik informacji o danej osobie."""  
    person = {'first': first_name, 'last': last_name} ①  
    return person ②  
  
musician = build_person('jimi', 'hendrix')  
print(musician) ③
```

---

Omawiana funkcja `build_person()` pobiera imię i nazwisko, a następnie te wartości umieszcza w słowniku (patrz wiersz ❶). Wartość argumentu `first_name` jest przechowywana wraz z kluczem 'first', natomiast wartość argumentu `last_name` wraz z kluczem 'last'. W wierszu ❷ polecenie `return` zwraca cały słownik przedstawiający daną osobę. Wartość zwrotna jest wyświetlana na ekranie w wierszu ❸, w którym pobieramy informacje tekstowe przechowywane w słowniku:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Powyzsza funkcja pobiera proste informacje tekstowe i umieszcza je w nieco bardziej skomplikowanej strukturze danych, która pozwala wykorzystać te informacje nie tylko w celu ich wyświetlania na ekranie. Ciągi tekstowe 'jimi' i 'hendrix' są teraz oznaczone jako imię i nazwisko. Tę funkcję można dalej rozbudować o przyjmowanie wartości dodatkowych, takich jak drugie imię, wiek, miejscowości i wszelkie inne informacje, które chcemy przechowywać o danej osobie. Przykładowo zmodyfikujmy teraz program tak, aby przechowywać również informacje o wieku osoby:

```
def build_person(first_name, last_name, age=None):
    """Zwraca słownik informacji o danej osobie."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

Do definicji funkcji dodaliśmy nowy parametr opcjonalny `age` i przypisaliśmy mu wartość specjalną `None`, która jest używana, gdy zmiennej nie zostanie przydana żadna konkretna wartość. `None` można potraktować jako miejsce zarezerwowane dla wartości. W wyrażeniu warunkowym `None` przyjmuje wartość `False`. Jeżeli wywołanie funkcji będzie zawierało wartość dla parametru `age`, zostanie ona umieszczona w słowniku. Omawiana funkcja zawsze przechowuje imię osoby, ale może zostać tak zmodyfikowana, aby przechowywać wszelkie inne informacje o danej osobie.

## Używanie funkcji wraz z pętlą while

Funkcji możesz używać razem ze wszystkimi poznanymi dotąd strukturami Pythona. Na przykład przedstawioną wcześniej funkcję `get_formatted_name()` wykorzystamy z pętlą `while` do bardziej formalnego powitania użytkowników. Poniżej przedstawiłem pierwszą próbę utworzenia kodu odpowiedzialnego za przywitanie użytkownika z wykorzystaniem jego imienia i nazwiska.

## Plik greeter.py:

---

```
def get_formatted_name(first_name, last_name):
    """Zwraca elegancko sformatowane pełne imię i nazwisko."""
    full_name = f'{first_name} {last_name}'
    return full_name.title()

# To jest pętla działająca w nieskończoność!
while True:
    print("\nProszę podać imię i nazwisko:") ❶
    f_name = input("Imię: ")
    l_name = input("Nazwisko: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nWitaj, {formatted_name}!")
```

---

W powyższym przykładzie użyliśmy prostej wersji funkcji `get_formatted_name()`, która nie zapewnia obsługi drugiego imienia. Pętla `while` prosi użytkownika o podanie pełnych danych, po czym umożliwia mu podanie oddzielnie imienia i nazwiska (patrz wiersz ❶).

Jednak mamy pewien problem z przedstawioną tutaj pętlą `while` — nie zdefiniowaliśmy warunku wyjścia. Powstaje w tym miejscu pytanie: gdzie należy umieścić warunek wyjścia, gdy program prosi użytkownika o podanie serii danych wejściowych? Chcemy umożliwić opuszczenie pętli najwcześniej, jak to możliwe, więc każde pytanie powinno oferować możliwość wyjścia. Polecamie `break` to najprostsze rozwiązanie, które pozwoli opuścić pętlę w trakcie wprowadzania poszczególnych danych wejściowych:

---

```
def get_formatted_name(first_name, last_name):
    """Zwraca elegancko sformatowane pełne imię i nazwisko."""
    full_name = f'{first_name} {last_name}'
    return full_name.title()

while True:
    print("\nProszę podać imię i nazwisko:")
    print("(wpisz 'q', aby zakończyć pracę w dowolnym momencie)")

    f_name = input("Imię: ")
    if f_name == 'q':
        break

    l_name = input("Nazwisko: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nWitaj, {formatted_name}!")
```

---

Dodaliśmy komunikat informujący użytkownika o możliwości wyjścia z programu w dowolnym momencie. Opuszczenie pętli nastąpi, gdy użytkownik wprowadzi wartość wyjścia zamiast danych wejściowych. Zwróć uwagę na to, że program będzie kontynuował witanie użytkowników dopóty, dopóki któryś z nich nie wprowadzi litery *q*:

---

Proszę podać imię i nazwisko:  
(wpisz 'q', aby zakończyć pracę w dowolnym momencie)  
Imię: **eryk**  
Nazwisko: **nowak**

Witaj, Eryk Nowak!

Proszę podać imię i nazwisko:  
(wpisz 'q', aby zakończyć pracę w dowolnym momencie)  
Imię: **q**

---

## ZRÓB TO SAM

**8.6. Nazwy miast.** Utwórz funkcję o nazwie `city_country()` pobierającą nazwę miasta i kraju, w którym ono leży. Wartością zwrotną funkcji powinien być ciąg tekstowy sformatowany w poniższy sposób:

---

Santiago, Chile

---

Przygotowaną funkcję wywołaj z przynajmniej trzema parami miasto-państwo i wyświetl wygenerowaną przez nie wartość.

**8.7. Album.** Utwórz funkcję o nazwie `make_album()` odpowiedzialną za zbudowanie słownika reprezentującego album muzyczny. Funkcja powinna pobrać nazwę zespołu lub artysty oraz tytuł albumu. Wartością zwrotną funkcji powinien być słownik zawierający te dwa fragmenty informacji. Za pomocą przygotowanej funkcji utwórz trzy słowniki przedstawiające różne albumy. Wyświetl każdą wartość zwrotną, aby pokazać, że słowniki prawidłowo przechowują informacje o albumach.

Wykorzystując wartość specjalną `None`, do funkcji `make_album()` dodaj opcjonalny parametr pozwalający na przechowywanie liczby utworów znajdujących się na płycie. Jeżeli wywołanie funkcji będzie zawierało wartość dla liczby utworów, należy ją dodać do słownika informacji o albumie. Zdefiniuj co najmniej jedno nowe wywołanie funkcji obejmujące także liczbę utworów na płycie.

**8.8. Albumy użytkowników.** Pracę rozpoczęj od programu utworzonego w ćwiczeniu 8.7. Dodaj pętlę `while` pozwalającą użytkownikom na wprowadzenie artysty i tytułu płyty. Po zebraniu tych informacji wywołaj funkcję `make_album()` wraz z podanymi przez użytkownika danymi wejściowymi oraz wyświetl słownik utworzony przez program. Upewnij się, że zdefiniowałeś wartość pozwalającą opuścić pętlę `while`.

# Przekazywanie listy

Bardzo często użyteczne będzie przekazywanie do funkcji listy, na przykład nazw, liczb lub bardziej skomplikowanych obiektów, takich jak słowniki. Kiedy zachodzi potrzeba przekazania listy do funkcji, wówczas ta funkcja otrzymuje bezpośredni dostęp do zawartości danej listy. Wykorzystamy teraz funkcje do znacznie efektywniejszej pracy z listami.

Przymajemy założenie, że mamy listę użytkowników i chcemy wyświetlać im spersonalizowane powitania. W przedstawionym poniżej przykładzie lista imion jest przekazywana do funkcji o nazwie `greet_users()`, która umożliwia indywidualne powitanie każdej osoby.

*Plik greet\_users.py:*

---

```
def greet_users(names):
    """Wyświetla proste powitanie każdemu użytkownikowi z listy."""
    for name in names:
        msg = f"Witaj, {name.title()}!"
        print(msg)

usernames = ['halina', 'tymek', 'marzena']
greet_users(usernames)
```

---

Zdefiniowaliśmy funkcję `greet_users()` w taki sposób, że oczekuje ona przekazania do niej listy imion, które będą przechowywane w parametrze `names`. Pętla funkcji przeprowadza iterację przez otrzymaną listę, a następnie wyświetla każdemu użytkownikowi komunikat powitania. Poza kodem funkcji zdefiniowaliśmy listę użytkowników (`usernames`), którą w następnym wierszu przekazaliśmy do funkcji `greet_users()`:

---

```
Witaj, Halina!
Witaj, Tymek!
Witaj, Marzena!
```

---

Otrzymane dane wyjściowe są zgodne z oczekiwaniemi. Każdy użytkownik jest witany w spersonalizowany sposób. Tę funkcję można wykorzystać zawsze, gdy trzeba będzie powitać określoną grupę użytkowników.

## Modyfikowanie listy w funkcji

Kiedy lista zostanie przekazana do funkcji, ta funkcja może ją zmodyfikować. Wszelkie zmiany wprowadzone na liście przez kod funkcji są trwałe. W ten sposób zyskujesz możliwość efektywnej pracy nawet podczas przetwarzania ogromnych ilości danych.

Rozważmy przykład firmy zajmującej się wydrukiem 3D modeli przekazywanych przez użytkowników. Projekty przeznaczone do wydruku są przechowywane

na liście, a po wydruku są przenoszone na zupełnie inną listę. Przedstawiony poniżej kod pokazuje rozwiążanie, które zostało opracowane bez użycia funkcji.

#### Plik printing\_models.py:

---

```
# Rozpoczynamy od pewnych projektów, które mają być wydrukowane.  
unprinted_designs = ['etui telefonu', 'robot pendant', 'dwunastościan']  
completed_models = []  
  
# Symulujemy wydruk poszczególnych projektów, dopóki pozostał jakikolwiek projekt  
# na liście. Każdy wydrukowany model zostaje przeniesiony na listę completed_models.  
while unprinted_designs:  
    current_design = unprinted_designs.pop()  
    print(f"Wydruk modelu: {current_design}")  
    completed_models.append(current_design)  
  
# Wyświetlenie wszystkich wydrukowanych modeli.  
print("\nWydrukowane zostały następujące modele:")  
for completed_model in completed_models:  
    print(completed_model)
```

---

Na początku programu definiujemy listę projektów przeznaczonych do wydruku oraz pustą listę o nazwie `completed_models`, na którą przeniesiony zostanie każdy wydrukowany model. Dopóki lista `unprinted_designs` zawiera jakieś kolwiek modele do wydrukowania, dopóty pętla `while` symuluje wydruk modelu: usuwa projekt znajdujący się na końcu listy, umieszcza go w zmiennej `current_design` oraz wyświetla komunikat o tym, że ten projekt jest aktualnie drukowany. W kolejnym kroku wydrukowany model zostaje przeniesiony na listę ukończonych modeli. Kiedy pętla zakończy działanie, lista wydrukowanych projektów zostanie wyświetlona na ekranie:

---

```
Wydruk modelu: dwunastościan  
Wydruk modelu: robot pendant  
Wydruk modelu: etui telefonu
```

```
Wydrukowane zostały następujące modele:  
dwunastościan  
robot pendant  
etui telefonu
```

---

Istnieje możliwość reorganizacji tego kodu przez utworzenie dwóch funkcji, z których każda stanie się odpowiedzialna za wykonanie określonego zadania. Większość kodu nie ulegnie zmianie, po prostu zapewnimy znacznie lepszą strukturę programu. Pierwsza funkcja będzie odpowiedzialna za obsługę wydruku projektu, natomiast druga wyświetli podsumowanie dotyczące wydrukowanych modeli:

---

```
def print_models(unprinted_designs, completed_models): ❶
    """
    Symulujemy wydruk poszczególnych projektów, dopóki pozostały jakikolwiek
    projekt na liście. Każdy wydrukowany model zostaje przeniesiony na
    listę completed_models.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Wydruk modelu: {current_design}")
        completed_models.append(current_design)

def show_completed_models(completed_models): ❷
    """Wyświetla wszystkie modele, które zostały wydrukowane."""
    print("\nWydrukowane zostały następujące modele:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['etui telefonu', 'robot pendant', 'dwunastościan']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

---

W wierszu ❶ definiujemy funkcję `print_models()` wraz z dwoma parametrami: listą projektów przeznaczonych do wydrukowania oraz listą wydrukowanych modeli. Mając do dyspozycji obie wymienione listy, funkcja symuluje przeprowadzenie wydruku poszczególnych projektów przez opróżnienie listy modeli przeznaczonych do wydruku i zapełnienie listy ukończonych zadań. Z kolei w wierszu ❷ definiujemy funkcję `show_completed_models()` wraz z jednym parametrem — listą wydrukowanych modeli. Na podstawie tej listy omawiana w tym przykładzie funkcja wyświetla nazwę każdego wydrukowanego modelu.

Zmodyfikowany program generuje takie same dane wyjściowe jak wersja pozbawiona funkcji, ale teraz kod jest znacznie bardziej zorganizowany. Kod odpowiedzialny za wykonanie większości pracy został przeniesiony do dwóch oddzielnych funkcji — dzięki ich istnieniu część głównego programu stała się łatwiejsza do zrozumienia. Spójrz na część główną programu i przekonaj się, że teraz znacznie łatwiej można określić, na czym polega działanie tego programu:

---

```
unprinted_designs = ['etui telefonu', 'robot pendant', 'dwunastościan']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

---

Definiujemy listę projektów przeznaczonych do wydruku oraz pustą listę przeznaczoną do umieszczania na niej już wydrukowanych modeli. Ponieważ mamy zdefiniowane dwie funkcje, następnym krokiem jest po prostu ich wywołanie

i przekazanie do nich prawidłowych argumentów. W wywołaniu funkcji `print_models()` podajemy obie wymagane przez nią listy, a `print_models()` zgodnie z naszymi oczekiwaniemi symuluje wydruk poszczególnych modeli. Następnie wywołujemy funkcję `show_completed_models()` i przekazujemy do niej listę wydrukowanych modeli, aby na tej podstawie mogła wygenerować podsumowanie dotyczące zrealizowanych projektów. Ponieważ nazwy funkcji wyraźnie wskazują ich przeznaczenie, inni programiści będą mogli łatwo odczytać kod i nawet bez komentarzy prawidłowo określić sposób jego działania.

Ten program jest łatwiejszy do obsługi i dalszej rozbudowy niż wersja niezawierająca funkcji. Jeżeli później zajdzie potrzeba wydrukowania kolejnych projektów, wystarczy, że po prostu ponownie wywoła się funkcję `print_models()`. Natomiast w przypadku, kiedy konieczne będzie zmodyfikowanie kodu odpowiedzialnego za symulację wydruku, zmianę trzeba będzie wprowadzić tylko w jednym miejscu, a zostanie uwzględniona wszędzie tam, gdzie wywołujemy funkcję `print_models()`. Takie podejście jest znacznie efektywniejsze niż konieczność uaktualniania kodu w wielu fragmentach programu.

Omówiony powyżej przykład wyraźnie pokazuje ideę, że każda funkcja powinna mieć do wykonania jedno konkretne zadanie. W tym przypadku pierwsza funkcja zajmuje się symulacją wydruku projektu, natomiast druga wyświetla informacje o wydrukowanych modelach. To jest znacznie lepsze rozwiązywanie niż użycie tylko jednej funkcji wykonującej oba zadania. Jeżeli tworzysz funkcję i zaczynasz zauważać, że wykonuje ona zbyt wiele różnych zadań, spróbuj rozdzielić kod na dwie różne funkcje. Pamiętaj o możliwości wywoływania jednej funkcji z poziomu innej, co okazuje się użyteczne podczas podziału skomplikowanych zadań na serię mniejszych kroków.

## Uniemożliwianie modyfikowania listy przez funkcję

Czasami chcesz uniemożliwić funkcji modyfikowanie listy. Przykładowo przyjmujemy założenie, że rozpoczęliśmy pracę z listą projektów przeznaczonych do wydruku i tworzymy funkcję przenoszącą ją na listę zrealizowanych zadań, podobnie jak miało to miejsce w poprzednim przykładzie. Jednak możemy postanowić, że nawet po wydrukowaniu wszystkich modeli chcemy zachować w archiwum wspomnianą listę projektów do wydrukowania. Ponieważ wszystkie nazwy projektów zostały przeniesione z listy `unprinted_designs`, lista ta jest teraz pusta i jest jedyną wersją, jaką dysponujesz. Pierwotna wersja listy została trwale utracona. Rozwiązaniem w omawianej sytuacji jest przekazywanie do funkcji kopii listy. Wszelkie zmiany wprowadzane przez funkcję będą miały wpływ jedynie na kopię listy, a nie na listę pierwotną, która w ten sposób pozostanie nietknęta.

Kopię listy można przekazać do funkcji w przedstawiony poniżej sposób:

---

```
nazwa_funkcji(nazwa_listy[:])
```

---

Notacja wycinka [:] powoduje utworzenie kopii listy przekazywanej do funkcji. Jeżeli w programie *print\_models.py* nie chcesz opróżnić listy projektów przeznaczonych do wydruku, wywołanie funkcji `print_models()` możesz zmienić na poniższe:

```
print_models(unprinted_designs[:], completed_models)
```

Funkcja `print_models()` nadal może wykonywać swoje zadanie, ponieważ wciąż otrzymuje nazwy wszystkich niewydrukowanych projektów. Jednak tym razem wykorzystujemy tylko kopię pierwotnej listy niewydrukowanych projektów, a nie rzeczywistą listę `unprinted_models`. Lista `completed_models` będzie zapelniania nazwami wydrukowanych modeli, podobnie jak miało to miejsce w poprzednim przykładzie. Natomiast pierwotna lista niewydrukowanych modeli pozostała niktnięta przez kod funkcji.

Wprawdzie zawartość listy można zachować przez przekazanie jej kopii do funkcji, ale mimo wszystko funkcjom należy przekazywać pierwotne listy, o ile nie istnieje ważny powód, aby to była kopia. Funkcja znacznie efektywniej pracuje z istniejącą listą, ponieważ unika poświęcania czasu i wykorzystania dodatkowej pamięci na utworzenie oddzielnej kopii listy. To ma szczególne znaczenie podczas pracy z ogromnymi listami.

## ZRÓB TO SAM

**8.9. Komunikaty.** Przygotuj listę zawierającą serię krótkich komunikatów, a następnie przekaż ją do funkcji o nazwie `show_messages()`, która powinna wyświetlić każdy komunikat umieszczony na tej liście.

**8.10. Wysyłanie komunikatów.** Pracę rozpocznij od kopii programu z ćwiczenia 8.9. Następnie utwórz funkcję o nazwie `send_messages()`, której zadaniem będzie wyświetlenie wszystkich komunikatów, a następnie przeniesienie ich na nową listę o nazwie `sent_messages`. Po wywołaniu funkcji należy wyświetlić obie listy i upewnić się o prawidłowym przeniesieniu komunikatów.

**8.11. Zarchiwizowane komunikaty.** Pracę rozpocznij od kodu utworzonego w ćwiczeniu 8.10. Wywołaj funkcję `send_messages()` wraz z kopią listy komunikatów. Po wywołaniu funkcji wyświetl obie listy, aby pokazać istnienie pierwotnej listy z zachowanymi komunikatami.

# Przekazywanie dowolnej liczby argumentów

Czasami wcześniej nie wiadomo, jaką liczbę argumentów będzie musiała akceptować funkcja. Na szczęście Python pozwala, aby funkcja pobierała dowolną liczbę argumentów z wywołującą ją polecenia.

Powróćmy na przykład do funkcji obsługującej zamówienia na pizzę. Tego rodzaju funkcja musi akceptować pewną liczbę dodatków, ale w trakcie projektowania

funkcji jeszcze nie wiemy, na ile dodatków zdecyduje się klient. W omawianym poniżej przykładzie funkcja ma tylko jeden parametr o nazwie `*toppings`, ale pobiera on dowolną liczbę argumentów dostarczanych przez polecenie wywołujące tę funkcję.

*Plik pizza.py:*

---

```
def make_pizza(*toppings):
    """Wyswietlenie listy dodatków wybranych przez klienta."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('pieczarki', 'zielona papryka', 'podwójny ser')
```

---

Gwiazdka w nazwie parametru `*toppings` informuje Pythona o konieczności utworzenia pustej krotki o nazwie `toppings` i umieszczenia w niej otrzymanych wartości. Polecenie `print()` w treści funkcji powoduje wygenerowanie danych wyjściowych pokazujących, że Python może obsługiwać wywołanie funkcji zawierające dowolną liczbę argumentów, na przykład jeden argument lub trzy. Poszczególne wywołania są traktowane podobnie. Zwrót uwagę, że argumenty zostają umieszczone w krotce, nawet jeśli funkcja otrzymuje tylko jedną wartość:

---

```
('pepperoni',)
('pieczarki', 'zielona papryka', 'podwójny ser')
```

---

W tym momencie polecenie `print()` możemy zastąpić pętlą przeprowadzającą iterację dodatków wybranych przez klienta i opisującą przygotowywaną pizzę:

---

```
def make_pizza(*toppings):
    """Podsumowanie informacji o przygotowywanej pizzy."""
    print("\nPrzygotowuję pizzę z następującymi dodatkami:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('pieczarki', 'zielona papryka', 'podwójny ser')
```

---

Wygenerowane dane wyjściowe pokazują, że działanie funkcji jest zgodne z oczekiwaniemi niezależnie od tego, czy przekazujemy jeden argument czy trzy:

---

```
Przygotowuję pizzę z następującymi dodatkami:
- pepperoni
```

```
Przygotowuję pizzę z następującymi dodatkami:
- pieczarki
- zielona papryka
- podwójny ser
```

---

Przygotowana składnia sprawdza się więc doskonale niezależnie od liczby argumentów przekazanych do funkcji.

## Argumenty pozycyjne i przekazywanie dowolnej liczby argumentów

Gdy chcesz, aby funkcja akceptowała wiele różnego rodzaju argumentów, wówczas parametr przyjmujący dowolną liczbę argumentów musi znajdować się na końcu definicji funkcji. Python najpierw dopasowuje argumenty pozycyjne oraz argumenty w postaci słów kluczowych, a dopiero później zbiera pozostałe argumenty dla ostatniego parametru.

Jeśli na przykład omawiana wcześniej funkcja musi uwzględnić również wielkość przygotowywanej pizzy, to parametr dotyczący wielkości trzeba umieścić przed parametrem `*toppings`:

---

```
def make_pizza(size, *toppings):
    """Podsumowanie informacji o przygotowanej pizzie."""
    print(f"\nPrzygotowuję pizzę o wielkości {size} cm, z następującymi dodatkami:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(40, 'pepperoni')
make_pizza(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

W powyższej definicji funkcji, pierwszą otrzymaną wartość Python przechowuje w parametrze `size`. Wszystkie pozostałe wartości zostają umieszczone w krotce `toppings`. W wywołaniu funkcji na początku podajemy wielkość przygotowywanej pizzy, a następnie dowolną liczbę wybranych do niej dodatków.

Teraz dane opisujące pizzę wskazują jej wielkość i wybrane dodatki, a poszczególne informacje są elegancko wyświetlane w odpowiednim miejscu. Najpierw wielkość pizzy, a dopiero później wybrane dodatki:

---

Przygotowuję pizzę o wielkości 40 cm, z następującymi dodatkami:  
- pepperoni

Przygotowuję pizzę o wielkości 30 cm, z następującymi dodatkami:  
- pieczarki  
- zielona papryka  
- podwójny ser

---

**UWAGA** *Bardzo często będziesz spotykać się z ogólną nazwą parametru `*args`, który zawiera dowolną liczbę argumentów pozycyjnych, takich jak użyte w omawianym przykładzie.*

## Używanie dowolnej liczby argumentów w postaci słów kluczowych

Czasami trzeba zaakceptować dowolną liczbę argumentów, choć wcześniej nie wiadomo, jakiego rodzaju informacje będą przekazywane do funkcji. W takim przypadku można przygotować funkcje akceptujące dowolną liczbę par klucz-wartość dostarczanych przez polecenie wywołujące tę funkcję. Jednym z przykładów może być tutaj budowa profilu użytkownika. Wprawdzie wiadomo, że otrzymamy informacje o użytkowniku, ale nie ma pewności, jakiego rodzaju to będą dane. Funkcja `build_profile()` w poniższym przykładzie zawsze pobiera imię i nazwisko, choć akceptuje również dowolną liczbę argumentów w postaci słów kluczowych.

Plik `user_profile.py`:

---

```
def build_profile(first, last, **user_info):
    """Budowa słownika zawierającego wszelkie informacje o użytkowniku."""
    user_info['first_name'] = first ❶
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='fizyka')
print(user_profile)
```

---

Definicja funkcji `build_profile()` oczekuje najpierw podania imienia i nazwiska, a dopiero później akceptuje dowolną liczbę par klucz-wartość. Dwie gwiazdki umieszczone w parametrze `**user_info` nakazują Pythonowi utworzenie pustego słownika o nazwie `user_info` oraz umieszczenie w nim wszystkich otrzymanych par klucz-wartość. Wewnątrz funkcji dostęp do tych par klucz-wartość przechowywanych w `user_info` można uzyskać dokładnie w taki sam sposób jak w przypadku każdego innego słownika.

W treści funkcji `build_profile()` dodajemy imię i nazwisko do słownika `profile`, ponieważ to zawsze będą dwa pierwsze fragmenty danych otrzymanych od użytkownika ❶, które jeszcze nie zostały umieszczone w słowniku. Na koniec słownik `user_info` jest wartością zwrotną funkcji `build_profile()`.

W przykładowym programie wywołujemy funkcję `build_profile()`, przekazując do niej imię 'albert', nazwisko 'einstein' oraz dwie pary klucz-wartość, czyli `location='princeton'` i `field='fizyka'`. Wartość zwrotna w postaci słownika `profile` zostaje umieszczona w zmiennej `user_profile`, której zawartość wyświetlamy na ekranie:

---

```
{'first_name': 'albert', 'last_name': 'einstein',
 'location': 'princeton', 'field': 'fizyka'}
```

---

Zwrócony przez funkcję słownik zawiera imię i nazwisko użytkownika oraz informacje dodatkowe — w omawianym przykładzie są nimi lokalizacja i dziedzina nauki, którą zajmuje się dana osoba. Przedstawiona funkcja będzie działała doskonale niezależnie od liczby par klucz-wartość dostarczanych przez poleceńie wywołujące tę funkcję.

Podczas tworzenia funkcji istnieje możliwość swobodnego łączenia argumentów pozycyjnych, argumentów w postaci słów kluczowych oraz dowolnego liczby pozostałych argumentów. Dobrze jest wiedzieć o istnieniu tych wszystkich typów argumentów, ponieważ będziesz się z nimi często spotykać, gdy zaczniesz analizować kod Pythona tworzony przez innych programistów. Umiejętność prawidłowego ustalania, kiedy i jak używać różnych typów argumentów, wymaga praktyki. Na razie zapamiętaj, aby stosować najprostsze podejście pozwalające na wykonanie zadania. Gdy zdobędziesz większą wiedzę i doświadczenie, wtedy nauczysz się stosować najbardziej efektywne rozwiązania.

**UWAGA** *Bardzo często będziesz spotykać się z parametrem `**kwargs`, który zawiera dowolną liczbę argumentów w postaci słów kluczowych.*

## ZRÓB TO SAM

**8.12. Kanapki.** Utwórz funkcję akceptującą listę składników, które klient chce umieścić w zamawianej kanapce. Funkcja powinna zawierać jeden parametr przechowujący dowolną liczbę argumentów przekazanych w wywołaniu funkcji oraz wyświetlać podsumowanie dotyczące zamówionej kanapki. Przygotowaną funkcję wywołaj trzykrotnie, za każdym razem z inną liczbą argumentów.

**8.13. Profil użytkownika.** Pracę rozpocznij od kopii programu `user_profile.py` utworzonego nieco wcześniej w tym rozdziale. Przygotuj własny profil przez wywołanie funkcji `build_profile()`, podaj imię, nazwisko oraz trzy inne pary klucz-wartość, które Cię opisują.

**8.14. Samochody.** Utwórz funkcję przechowującą w słowniku informacje o samochodzie. Ta funkcja zawsze powinna otrzymywać nazwy marki i modelu pojazdu, po którym można podać dowolną liczbę argumentów w postaci słów kluczowych. Wywołaj funkcję zawierającą wymagane informacje oraz dwie dodatkowe pary nazwa-wartość, na przykład opisujące kolor i wyposażenie dodatkowe. Przygotowana funkcja powinna być wywoływana w sposób podobny do przedstawionego poniżej:

---

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

---

Wyświetl zawartość słownika zwróconego przez tę funkcję i upewnij się, że wszystkie podane informacje zostały w nim prawidłowo zapisane.

# Przechowywanie funkcji w modułach

Jedną z zalet funkcji jest możliwość oddzielenia bloków kodu od części głównej programu. Dzięki użyciu opisowych nazw dla funkcji poprawne ustalenie sposobu działania programu stanie się łatwiejsze. Można pójść jeszcze o krok dalej umieścić funkcje w oddzielnym pliku określonym mianem *modułu*, a następnie importować ten moduł do programu, w którym mają być wywoływanie funkcje danego modułu. Polecenie `import` nakazuje Pythonowi, aby kod znajdujący się w module udostępnił aktualnie wykonywanemu plikowi programu.

Przechowywanie funkcji w oddzielnym programie pozwala ukryć szczegóły kodu programu i skoncentrować się na logice wyższego poziomu. Ponadto tak przygotowane funkcje będą mogły być wielokrotnie używane także w innych programach. Kiedy przechowujesz funkcje w oddzielnych plikach, mogą być one udostępniane innym programistom bez konieczności udostępniania całego programu. Umiejętność i możliwość importowania funkcji pozwala Ci również używać bibliotek funkcji opracowanych przez innych programistów.

Istnieje wiele sposobów na import modułu, poniżej przedstawiłem ich krótkie omówienie.

## Import całego modułu

Aby rozpocząć importowanie funkcji, najpierw trzeba zacząć od utworzenia modułu. Wspomniany *moduł* to plik o rozszerzeniu `.py` zawierający kod, który będzie później importowany do programu. Przystępujemy teraz do utworzenia modułu udostępniającego funkcję `make_pizza()`. Pracę nad modelem zaczynamy od usunięcia z pliku `pizza.py` całego kodu poza funkcją `make_pizza()`.

Plik `pizza.py`:

---

```
def make_pizza(size, *toppings):
    """Podsumowanie informacji o przygotowywanej pizzy."""
    print(f"\nPrzygotowuję pizzę o wielkości {size} cm, z następującymi
dodatkami:")
    for topping in toppings:
        print(f"- {topping}")
```

---

Następnym krokiem jest utworzenie oddzielnego pliku o nazwie *making\_pizzas.py*, który należy umieścić w katalogu zawierającym plik `pizza.py`. Nowy plik będzie importował utworzony przed chwilą moduł i wykonywał dwa wywołania funkcji `make_pizza()`.

Plik `making_pizzas.py`:

---

```
import pizza

pizza.make_pizza(40, 'pepperoni') ❶
pizza.make_pizza(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

Kiedy Python odczyta ten plik, polecenie `import pizza` spowoduje otworzenie pliku `pizza.py` i skopiowanie wszystkich znajdujących się w nim funkcji do bieżącego programu. W rzeczywistości nie widać kodu skopowanego między plikami, ponieważ operacja kopирования jest przez Pythona przeprowadzana w tle, podczas działania programu. Musisz jedynie wiedzieć, że wszystkie funkcje zdefiniowane w pliku `pizza.py` będą dostępne w programie `making_pizzas.py`.

Aby wywołać funkcję z zainportowanego modułu, należy podać jego nazwę (tutaj `pizza`), a po kropce nazwę funkcji, na przykład `make_pizza()`, tak jak pokazałem w wierszu ❶. Ten kod spowoduje wygenerowanie takich samych danych wyjściowych jak w przypadku pierwotnego programu, którego działanie nie opierało się na module:

---

Przygotowuję pizzę o wielkości 40 cm, z następującymi dodatkami:  
- pepperoni

Przygotowuję pizzę o wielkości 30 cm, z następującymi dodatkami:  
- pieczarki  
- zielona papryka  
- podwójny ser

---

To jest pierwsze podejście, jeśli chodzi o sposoby importowania modułu. Polega ono po prostu na użyciu polecenia `import` i podaniu nazwy modułu. W ten sposób wszystkie funkcje znajdujące się w danym module zostaną udostępnione bieżącemu programowi. Jeżeli użyjesz tego rodzaju polecenia `import` do zainportowania całego modułu o nazwie `nazwa_modułu.py`, wówczas poszczególne funkcje tego modułu staną się dostępne za pomocą poniższej składni:

---

`nazwa_modułu.nazwa_funkcji()`

---

## Import określonych funkcji

Istnieje również możliwość zainportowania jedynie określonych funkcji z modułu. Poniżej przedstawiłem ogólną składnię stosowaną w takim podejściu:

---

`from nazwa_modułu import nazwa_funkcji`

---

Z modułu można zainportować dowolną liczbę funkcji, których nazwy powinny być rozdzielone przecinkami, tak jak pokazałem poniżej:

---

`from nazwa_modułu import nazwa_funkcji_0, nazwa_funkcji_1,  
 nazwa_funkcji_2`

---

Powracamy do naszego programu *making\_pizzas.py*. Kiedy chcemy zaimportować jedynie funkcję, której będziemy używać, wtedy polecenie `import` będzie miało następującą postać:

---

```
from pizza import make_pizza

make_pizza(40, 'pepperoni')
make_pizza(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

W przypadku tej składni nie trzeba używać notacji z kropką podczas wywoływania funkcji. Ponieważ w poleceniu `import` wyraźnie zaimportowaliśmy funkcję `make_pizza()`, wywołujemy ją za pomocą nazwy, podobnie jak używamy każdej innej funkcji.

## Użycie słowa kluczowego `as` w celu zdefiniowania aliasu funkcji

Gdy nazwa importowanej funkcji może kolidować z nazwą funkcji istniejącej już w programie, lub też ta nazwa jest zbyt dłużna, wówczas można zastosować skrót. Wspomniany skrót jest unikatowym *aliasem*, czyli nazwą alternatywną, podobnie jak nick dla funkcji. Ten specjalny nick nadajemy funkcji podczas jej importu.

W poniższym fragmencie kodu dla funkcji `make_pizza()` definiujemy alias `mp()`, co odbywa się przez użycie zapisu `make_pizza as mp`. Słowo kluczowe `as` zmienia nazwę funkcji, używając podanego aliasu:

---

```
from pizza import make_pizza as mp

mp(40, 'pepperoni')
mp(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

Przedstawione powyżej polecenie import zmienia w bieżącym programie nazwę funkcji `make_pizza()` na `mp()`. Za każdym razem, gdy chcesz wywołać `make_pizza()`, możesz po prostu użyć wywołania `mp()`, a Python wykona kod zdefiniowany w funkcji `make_pizza()`. W ten sposób można uniknąć konfliktu z inną funkcją o nazwie `make_pizza()`, która mogłaby się znajdować w bieżącym pliku programu.

Ogólna składnia pozwalająca utworzyć alias przedstawia się następująco:

---

```
from nazwa_modułu import nazwa_funkcji as alias
```

---

## Użycie słowa kluczowego as w celu zdefiniowania aliasu modułu

Istnieje również możliwość utworzenia aliasu dla nazwy modułu. Nadanie modułowi krótkiego aliasu, na przykład `p` dla `pizza`, pozwala na jeszcze szybsze wywoływanie funkcji modułu. Dlatego też wywołanie `p.make_pizza()` będzie znacznie zwięzlejsze niż `pizza.make_pizza()`:

---

```
import pizza as p

p.make_pizza(40, 'pepperoni')
p.make_pizza(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

W poleceniu `import` dla modułu `pizza` tworzymy alias `p`, natomiast nazwy wszystkich funkcji modułu pozostają niezmienione. Wywołanie funkcji za pomocą `p.make_pizza()` jest nie tylko zwięzlejsze niż zapis `pizza.make_pizza()`, lecz także odciąga Twoją uwagę od nazwy modułu i pozwala skoncentrować się na jasnych i czytelnych nazwach jego funkcji. Wspomniane nazwy funkcji, które jasno i wyraźnie wskazują ich przeznaczenie, są z perspektywy czytelności kodu znacznie ważniejsze niż użycie pełnej nazwy modułu.

Ogólna składnia pozwalająca utworzyć alias dla modułu przedstawia się następująco:

---

```
import nazwa_modułu as alias
```

---

## Import wszystkich funkcji modułu

Za pomocą operatora `*` można nakazać Pythonowi zimportowanie wszystkich funkcji znajdujących się w module:

---

```
from pizza import *

make_pizza(40, 'pepperoni')
make_pizza(30, 'pieczarki', 'zielona papryka', 'podwójny ser')
```

---

W powyższym fragmencie kodu gwiazdka w poleceniu `import` nakazuje Pythonowi skopiowanie każdej funkcji z modułu `pizza` do pliku bieżącego programu. Ponieważ zimportowane zostają wszystkie funkcje, można je wywoływać bez konieczności użycia notacji z kropką. Jednak podczas pracy z ogromnymi modułami opracowanymi przez innych programistów lepiej jest nie stosować tego podejścia, ponieważ gdy moduł zawiera funkcję o nazwie takiej samej jak nazwa funkcji istniejącej już w projekcie, wówczas mogą wystąpić nieoczekiwane efekty. Python może rozpoznawać wiele funkcji lub zmiennych o tej samej nazwie i zamiast je wszystkie oddziennie zaimportować, zacznie po prostu nadpisywać funkcje.

Najlepsze podejście polega na zimportowaniu funkcji, których chcesz używać, lub całego modułu, a następnie na stosowaniu notacji z kropką. W ten sposób powstaje czytelny kod, który będzie jasny i zrozumiały dla innych programistów. O poleceniu import występującym razem z gwiazdką wspomniałem tylko dlatego, że możesz się spotkać z takim rozwiążaniem podczas analizy kodu utworzonego przez innych programistów:

---

```
from nazwa_modułu import *
```

---

## Nadawanie stylu funkcjom

Podczas nadawania stylu funkcjom powinieneś pamiętać o kilku kwestiach. Nazwa funkcji powinna jasno sugerować jej przeznaczenie oraz składać się z małych liter i znaków podkreślenia. Tego rodzaju opisowa nazwa pomoże zarówno Tobie, jak i innym programistom w ustaleniu przeznaczenia danego fragmentu kodu. Dla nazw modułów również należy stosować tego rodzaju konwencję.

W kodzie każdej funkcji powinien znajdować się komentarz zwięzły wyjaśniający przeznaczenie danej funkcji. Ten komentarz należy umieścić tuż po definicji funkcji i zastosować dla niego format *docstring*. W przypadku doskonale udokumentowanej funkcji inni programiści mogą jej używać po zapoznaniu się jedynie z komentarzem umieszczonym w *docstring*. Gdy ufają, że kod działa zgodnie z przedstawionym opisem, i gdy znają nazwę funkcji, wymagane argumenty oraz rodzaj wartości zwrotnej, wówczas mogą wykorzystać tę funkcję we własnych programach.

Jeżeli podajesz wartość domyślną dla parametru, po obu stronach znaku równości nie należy umieszczać żadnych spacji:

---

```
def nazwa_funkcji(parametr_0, parametr_1='wartość domyślna')
```

---

Ta sama konwencja powinna być używana względem argumentów w postaci słów kluczowych stosowanych w wywołaniach funkcji:

---

```
nazwa_funkcji(wartość_0, parametr_1='wartość')
```

---

Specyfikacja PEP 8 (<https://www.python.org/dev/peps/pep-0008/>) zaleca ograniczenie długości wiersza kodu do 79 znaków, aby każdy wiersz był widoczny w oknie edytora na ekranie o rozsądnej wielkości. Gdy parametry powodują, że długość definicji funkcji wykracza poza wspomniane 79 znaków, wówczas po nawiasie otwierającym definicję naciśnij klawisz *Enter*. W następnym wierszu dwukrotnie naciśnij klawisz tabulatora, aby oddzielić listę argumentów od treści funkcji, która będzie wcięta tylko o jeden poziom.

Większość edytorów automatycznie stosuje wcięcia dla dodatkowych wierszy parametrów, aby dopasować je do wcięcia zastosowanego w pierwszym wierszu definicji:

---

```
def nazwa_funkcji(  
    parametr_0, parametr_1, parametr_2,  
    parametr_3, parametr_4, parametr_5):  
    treść funkcji...
```

---

Jeżeli program lub moduł zawiera więcej niż tylko jedną funkcję, dozwolone jest ich rozdzielenie dwoma pustymi wierszami, co pozwala łatwiej dostrzec miejsce zakończenia jednej funkcji i rozpoczęcia następnej.

Wszystkie polecenia `import` powinny znajdować się na początku pliku. Jedynym wyjątkiem jest sytuacja, gdy na początku pliku umieszczasz komentarze opisujące ogólne przeznaczenie i działanie programu.

## ZRÓB TO SAM

**8.15. Wydruk modeli.** Funkcje z programu `print_models.py` umieść w oddzielnym pliku o nazwie `printing_functions.py`. Na początku pliku `print_models.py` umieść polecenie `import` i zmodyfikuj plik w taki sposób, aby używać zaimportowanych funkcji.

**8.16. Polecenia importu.** Wykorzystaj utworzony przez siebie program z jedną funkcją i przenieś ją do oddzielnego pliku. Zainportuj tę funkcję w pliku programu głównego, a następnie wywołaj funkcję na wszystkie wymienione poniżej sposoby:

---

```
import nazwa_modułu  
from nazwa_modułu import nazwa_funkcji  
from nazwa_modułu import nazwa_funkcji as fn  
import nazwa_modułu as mn  
from nazwa_modułu import *
```

---

**8.17. Nadanie stylu funkcjom.** Wybierz trzy dowolne programy utworzone w tym rozdziale i upewnij się, że są w nich stosowane przedstawione w tym podrozdziale konwencje dotyczące nadawania stylu funkcjom.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak tworzyć funkcje i przekazywać do nich argumenty, aby funkcje te mogły uzyskać dostęp do informacji niezbędnych podczas wykonywania zadań, do których zostały przeznaczone. Zobaczyłeś, jak można wykorzystać argumenty pozycyjne oraz argumenty w postaci słów kluczowych, a także jak akceptować dowolną liczbę argumentów. Poznałeś również

funkcje wyświetlające dane wyjściowe oraz zwracające wartość. Dowiedziałeś się, jak używać funkcji razem z listami, słownikami, poleceniami `if` oraz pętlami `while`. Zobaczyłeś, jak przechowywać funkcje w oddzielnych plikach nazywanych *modułami*, aby pliki główne programów stały się prostsze i łatwiejsze do zrozumienia. Na końcu poznałeś zalecenia dotyczące nadawania stylu funkcjom, dzięki którym programy nadal będą charakteryzowały się doskonałą strukturą oraz będą łatwiejsze do odczytania zarówno dla Ciebie, jak i dla innych programistów.

Jednym z celów programisty powinno być tworzenie prostego kodu wykonującego zadanie, do którego został on przeznaczony. Funkcje pomagają w osiągnięciu tego celu. Umożliwiają przygotowanie bloków kodu i pozostawienie ich w spokoju, gdy wiadomo, że działanie funkcji jest zgodne z oczekiwaniemi. Kiedy masz świadomość prawidłowego wykonywania przez funkcję zadania, do którego została przeznaczona, możesz przyjąć założenie, że będzie ona nadal działała i spokojnie przejść do następnego zadania programistycznego.

Funkcje pozwalają raz utworzonego kodu używać wielokrotne. Kiedy chcesz uruchomić kod zdefiniowany w funkcji, Twoje zadanie sprowadza się zaledwie do przygotowania jednowierszowego wywołania, a wskazana w nim funkcja wykona swoje zadanie. W przypadku kiedy będzie konieczne zmodyfikowanie zachowania funkcji, zmiany trzeba będzie wprowadzić tylko w jednym bloku kodu, aby zostały one zastosowane we wszystkich wywołaniach tej funkcji.

Użycie funkcji powoduje, że tworzone na ich bazie programy stają się łatwiejsze do odczytania, a dobre nazwy funkcji jasno i wyraźnie podsumowują działanie danego fragmentu kodu. Odczyt serii wywołań funkcji pozwoli na znacznie szybsze określenie sposobu działania programu niż przypadku analizy długich serii bloków kodu.

Dzięki funkcjom łatwiejsze staje się również testowanie i debugowanie kodu źródłowego. Kiedy większa część pracy programu jest wykonywana przez zbiór funkcji, z których każda ma określone zadanie, wówczas znacznie łatwiej można testować i obsługiwać tak opracowany kod źródłowy. Możesz utworzyć oddzielny program wywołujący każdą funkcję i sprawdzający jej działanie we wszystkich możliwych sytuacjach. Kiedy zdecydujesz się na takie podejście, możesz mieć pewność, że funkcje będą działały prawidłowo w trakcie ich każdego wywołania.

W rozdziale 9. dowiesz się, jak projektować klasy. Wspomniane *klasy* łączą funkcje i dane, tworząc rodzaj eleganckiego pakietu, który może być używany w sposób niezwykle elastyczny i efektywny.

# 9

## Klasy



**PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO TO JEDNO Z NAJEFEKTYWNIEJSZYCH PODEJŚĆ PODCZAS TWORZENIA OPROGRAMOWANIA.**

W PRZYPADKU PROGRAMOWANIA ZORIENTOWANEGO OBIEKTOWO PRZYGOTOWUJEMY *klasy* reprezentujące rzeczywiste rzeczy i sytuacje, a następnie tworzymy *obiekty* na podstawie tych klas. Kiedy opracowujesz klasę, definiujesz ogólne zachowanie dla danej kategorii obiektów. Natomiast gdy tworzyś poszczególne obiekty na bazie klas, każdy z tych obiektów automatycznie otrzymuje ogólnie zachowanie, choć można mu również przypisać unikatowe cechy, które będą go odróżniać od pozostałych obiektów. Będziesz zaskoczony tym, ile rzeczywistych sytuacji można modelować za pomocą programowania zorientowanego obiektowo.

Budowanie obiektu na podstawie klasy nosi nazwę *tworzenia egzemplarza* — później w kodzie pracujesz z *egzemplarzami* klas. W tym rozdziale będziemy definiować klasy i tworzyć ich egzemplarze. Określisz rodzaj informacji, jakie mogą być przechowywane w egzemplarzach, a następnie zdefiniujesz akcje, które będą podejmowane w tych egzemplarzach. Ponadto przygotujesz klasy rozszerzające funkcjonalności tych już istniejących, aby podobne klasy mogły efektywnie współdzielić kod. Opracowane przez siebie klasy umieścisz w modułach, a klasy utworzone przez innych programistów będziesz importować w plikach własnych programów.

Zrozumienie programowania zorientowanego obiektowo pomoże Ci spojrzeć na świat z perspektywy programisty. Jeszcze lepiej poznasz kod — nie tylko sposób jego działania wiersz po wierszu, lecz także ogólne koncepcje stojące za nim. Poznanie logiki związanej z klasami pozwoli Ci zastosować logiczne podejście

podczas tworzenia programów, które będą efektywnie rozwiązywać praktycznie wszystkie napotykane przez Ciebie problemy.

Gdy wyzwania i projekty do realizacji stają się większe oraz bardziej skomplikowane, klasy ułatwiają pracę zarówno Tobie, jak i innym programistom, z którymi współpracujesz. Kiedy programiści tworzą kod oparty na tej samej logice, są w stanie zrozumieć sposób działania kodu napisanego przez inną osobę. Przygotowane przez Ciebie fragmenty kodu będą miały sens dla pozostałych osób w zespole i każda z nich będzie mogła dodać do tego kodu jeszcze coś od siebie.

## Utworzenie i użycie klasy

Z pomocą klas można modelować praktycznie wszystko. Zaczynamy od utworzenia prostej klasy `Dog` przedstawiającej psa — nie chodzi tutaj o konkretnego psa, ale ogólnie o każdego psa. Co wiemy o większości psów? Cóż, wszystkie mają swój wiek oraz imię. Ponadto większość psów potrafi siedzieć i kłaść się na plecach. Te dwa rodzaje informacji (imię i wiek) oraz dwa zachowania (siedzenie i kładzenie się na plecach) zdefiniujemy w przygotowywanej klasie `Dog`, ponieważ to są cechy wspólne większości psów. Opracowana przez nas klasa wskaże Pythonowi, jak zdefiniować obiekt przedstawiający psa. Po przygotowaniu klasy wykorzystamy ją do utworzenia poszczególnych egzemplarzy, z których każdy będzie przedstawiać jednego konkretnego psa.

### Utworzenie klasy `Dog`

Każdy egzemplarz utworzony na podstawie klasy `Dog` będzie przechowywał informacje o imieniu (`name`) i wieku (`age`), a także zyska możliwość siedzenia (`sit()`) i kładzenia się na plecach (`roll_over()`).

Plik `dog.py`:

---

```
class Dog: ❶
    """Prosta próba modelowania psa."""

    def __init__(self, name, age): ❷
        """Inicjalizacja atrybutów name i age."""
        self.name = name ❸
        self.age = age

    def sit(self): ❹
        """Symulacja, że pies siada po otrzymaniu polecenia."""
        print(f"{self.name.title()} teraz siedzi.")

    def roll_over(self):
        """Symulacja, że pies kładzie się na plecy po otrzymaniu polecenia."""
        print(f"{self.name.title()} teraz położył się na plecy!")
```

---

W powyższym fragmencie kodu wiele się dzieje, ale nie przejmuj się tym. Tę samą strukturę zobaczysz wielokrotnie w rozdziale i będziesz miał jeszcze sporo czasu na oswojenie się z nią. W wierszu ❶ definiujemy klasę o nazwie `Dog`. Zgodnie z konwencją nazwa klasy w Pythonie rozpoczyna się od wielkiej litery. Definicja klasy nie ma nawiasu, ponieważ tworzymy ją zupełnie od początku. Następnie mamy komentarz w stylu *docstring* opisujący przeznaczenie klasy.

## Metoda `__init__()`

Funkcja będąca częścią klasy nosi nazwę *metody*. Wszystko to, czego dotąd nauczyłeś się o funkcjach, ma zastosowanie także względem metod. Jedyna praktyczna różnica polega na tym, że metody są wywoływanne w inny sposób. Przedstawiona w wierszu ❷ metoda `__init__()` jest zaliczana do metod specjalnych i będzie przez Pythona wywołana automatycznie w trakcie każdej operacji tworzenia nowego egzemplarza na podstawie klasy `Dog`. W nazwie tej metody znajdują się dwa znaki podkreślenia na początku i na końcu — jest to konwencja zapisu stosowana przez Pythona w celu uniknięcia sytuacji, w której nazwy metod domyślnych będą kolidowały z metodami opracowanymi przez programistów. Upewnij się, że po obu stronach słowa `init` w nazwie metody `__init__()` zostały umieszczone dwa znaki podkreślenia. Jeżeli użyjesz tylko po jednym znaku podkreślenia, ta metoda nie zostanie wywołana automatycznie podczas tworzenia egzemplarza klasy, co może prowadzić do błędów, które będą trudne do wychwycenia.

Zdefiniowana tutaj metoda `__init__()` ma trzy parametry: `self`, `name` i `age`. Parametr o nazwie `self` jest wymagany w definicji metody, a na dodatek musi znajdować się przed pozostałymi parametrami. Jego obecność w definicji jest niezbędna, ponieważ kiedy Python wywoła tę metodę `__init__()`, aby utworzyć egzemplarz klasy `Dog`, wówczas wywołanie metody automatycznie przekaże argument `self`. Każde wywołanie metody powiązane z klasą automatycznie przekazuje argument `self`, który jest odwołaniem do danego egzemplarza. Dzięki temu poszczególne egzemplarze uzyskują dostęp do atrybutów i metod zdefiniowanych w klasie. Kiedy tworzymy egzemplarz klasy `Dog`, Python wywoła metodę `__init__()` z klasy `Dog`. W trakcie tego wywołania przekazane zostają argumenty `name` i `age`, natomiast argument `self`, jak już wcześniej wspomniałem, zostanie przekazany automatycznie, więc nie będzie musiał być przekazywany ręcznie. Gdy zajdzie potrzeba utworzenia egzemplarza na podstawie klasy `Dog`, konieczne będzie dostarczenie wartości jedynie dla dwóch ostatnich parametrów — `name` i `age`.

Dwie zmienne zdefiniowane w wierszu ❸ i poniższym mają prefiks `self`. Jeżeli zmienna ma wymieniony prefiks, jest dostępna dla każdej metody w klasie. Ponadto uzyskanie dostępu do tych zmiennych jest możliwe za pomocą dowolnego egzemplarza utworzonego na bazie danej klasy. Polecenie `self.name = name` pobiera wartość przechowywaną w parametrze `name` i umieszcza ją w zmiennej `name`, która następnie zostaje dołączona do tworzonego egzemplarza. Ten sam proces zachodzi w przypadku `self.age = age`. Zmienne dostępne za pomocą egzemplarzy takich jak omawiany są nazywane *atrybutami*.

W klasie Dog mamy zdefiniowane jeszcze dwie inne metody: `sit()` i `roll_over()`, jak możesz zobaczyć w wierszu ④. Ponieważ te metody nie wymagają żadnych informacji dodatkowych, takich jak imię psa lub jego wiek, to definiujemy je z jednym parametrem — `self`. Utworzone później egzemplarze będą miały dostęp także do tych metod dodatkowych. Innymi słowy: w egzemplarzach utworzonych na bazie klasy Dog będzie możliwa wywoływać metody `sit()` i `roll_over()`. Na razie działanie tych metod nie jest zbyt ekscytujące — po prostu wyświetlają komunikat o tym, że pies siedzi lub położył się na plecach. Jednak tę koncepcję można rozbudować i dostosować do bardziej rzeczywistych sytuacji. Jeżeli klasa byłaby częścią gry komputerowej, wymienione metody mogłyby zawierać kod tworzący animację siadającego lub kładącego się na plecach psa. Natomiast jeśli klasa zostałaby napisana w celu kontrolowania robota, te metody bezpośrednio kierowałyby robotem w postaci psa, aby ten siadał lub kłał się na plecach.

## Utworzenie egzemplarza na podstawie klasy

Potraktuj klasę jako zbiór instrukcji wskazujących sposób zbudowania egzemplarza. Klasa Dog będzie więc zbiorem instrukcji wskazujących Pythonowi, jak można utworzyć poszczególne egzemplarze reprezentujące konkretne psy.

Przystępujemy do zbudowania egzemplarza konkretnego psa:

---

```
class Dog:  
    --cięcie--  
  
my_dog = Dog('willie', 6) ①  
  
print(f'Mój pies ma na imię {my_dog.name.title()}') ②  
print(f'Mój pies ma {my_dog.age} lat.') ③
```

---

W powyższym fragmencie kodu używamy klasy `Dog`, którą przygotowaliśmy we wcześniejszym przykładzie. W wierszu ① nakazujemy Pythonowi utworzenie egzemplarza reprezentującego psa, który ma na imię 'willie' i ma 6 lat. Kiedy Python odczytuje ten wiersz kodu, wywołuje metodę `__init__()` klasy `Dog` wraz z argumentami 'willie' i 6. Metoda `__init__()` tworzy egzemplarz przedstawiający tego konkretnego psa oraz przypisuje wartości atrybutom `name` i `age` na podstawie wartości dostarczonych w wywołaniu. W metodzie `__init__()` nie mamy wyraźnie zdefiniowanego polecenia `return`, ale Python automatycznie zwraca egzemplarz reprezentujący danego psa. Ten egzemplarz zostaje umieszczony w zmiennej `my_dog`. Konwencja nazewnica okazuje się tutaj przydatna. Zwykle można przyjąć założenie, że nazwa rozpoczęjąca się od wielkiej litery, taka jak `Dog`, odwołuje do klasy, natomiast nazwa zapisana małymi literami, taka jak `my_dog`, odwołuje do pojedynczego egzemplarza utworzonego na podstawie tej klasy.

## Uzyskanie dostępu do atrybutów

W celu uzyskania dostępu do atrybutów egzemplarza używamy notacji z kropką. W wierszu ❷ uzyskujemy dostęp do wartości atrybutu `name` w egzemplarzu `my_dog` za pomocą przedstawionego poniżej polecenia:

---

```
my_dog.name
```

---

Notacja z kropką jest często używana w Pythonie. Tego rodzaju składnia pokazuje, jak Python znajduje wartość atrybutu. W omawianym przykładzie Python szuka egzemplarza `my_dog`, a następnie powiązanego z nim atrybutu `name`. To jest dokładnie ten sam atrybut, który w klasie `Dog` jest wskazywany przez polecenie `self.name`. W wierszu ❸ wykorzystujemy takie samo podejście do pracy z atrybutem `age`.

Wygenerowane dane wyjściowe zawierają informacje zebrane o danym egzemplarzu `my_dog`:

---

```
Mój pies ma na imię Willie.  
Mój pies ma 6 lat.
```

---

## Wywoływanie metod

Kiedy utworzymy już egzemplarz na podstawie klasy `Dog`, notacji z kropką możemy użyć do wywołania dowolnej metody zdefiniowanej w tej klasie. Teraz skorzystamy z tej możliwości, aby kazać psu usiąść, a później położyć się na plecach:

---

```
class Dog:  
    --cięcie--  
  
    my_dog = Dog('willie', 6)  
    my_dog.sit()  
    my_dog.roll_over()
```

---

W celu wywołania metody podajemy nazwę egzemplarza (w omawianym przykładzie to `my_dog`) oraz metody, która ma być wywołana. Wymienione komponenty rozdzielimy kropką. Kiedy Python odczyta polecenie `my_dog.sit()`, odszuka metodę `sit()` w klasie `Dog` i wykona zdefiniowany w niej kod. Interpretacja polecenia `my_dog.roll_over()` odbywa się w dokładnie taki sam sposób.

Teraz nasz pies Willie robi to, o co go prosimy:

---

```
Willie teraz siedzi.  
Willie teraz położył się na plecach!
```

---

Zastosowana tutaj składnia jest całkiem użyteczna. Kiedy atrybuty i metody mają wystarczająco jasne i czytelne nazwy, takie jak `name`, `age`, `sit()` i `roll_over()`, wówczas możemy bardzo łatwo określić przeznaczenie danego bloku kodu, nawet jeśli nigdy wcześniej nie widzieliśmy jego zawartości.

## Utworzenie wielu egzemplarzy

Można utworzyć dowolną liczbę egzemplarzy na podstawie jednej klasy. Przy-stępujemy teraz do utworzenia egzemplarza o nazwie `your_dog`, który będzie reprezentował drugiego psa:

---

```
class Dog:  
    --cięcie--  
  
    my_dog = Dog('willie', 6)  
    your_dog = Dog('lucy', 5)  
  
    print(f"\nMój pies ma na imię {my_dog.name.title()}.")  
    print(f"Twój pies ma {my_dog.age} lat.")  
    my_dog.sit()  
  
    print(f"\nTwój pies ma na imię {your_dog.name.title()}.")  
    print(f"Twój pies ma {your_dog.age} lat.")  
    your_dog.sit()
```

---

W powyższym fragmencie kodu utworzyliśmy egzemplarze reprezentujące dwa psy: jednego o imieniu Willie i drugiego o imieniu Lucy. Każdy pies jest oddzielnym egzemplarzem, który posiada własny zbiór atrybutów, ale jednocześnie oba psy mogą podjąć dokładnie te same akcje:

---

```
Mój pies ma na imię Willie.  
Mój pies ma 6 lat.  
Willie teraz siedzi.
```

```
Twój pies ma na imię Lucy.  
Twój pies ma 5 lat.  
Lucy teraz siedzi.
```

---

Nawet jeśli użylibyśmy tego samego imienia i wieku dla drugiego psa, Python i tak utworzyłby oddzielne egzemplarze na bazie klasy `Dog`. Na podstawie pojęcia klasy można utworzyć dowolną liczbę egzemplarzy, jeżeli każdy z nich będzie miał unikatową nazwę zmiennej lub zajmował inne miejsce na liście bądź w słowniku.

## ZRÓB TO SAM

**9.1. Restauracja.** Przygotuj klasę o nazwie Restaurant. Metoda `__init__()` w klasie Restaurant powinna przechowywać dwa atrybuty: `restaurant_name` i `cuisine_type`. Utwórz metodę o nazwie `describe_restaurant()` wyświetlającą te dwa fragmenty informacji oraz metodę o nazwie `open_restaurant()` wyświetlającą informacje o godzinach pracy restauracji.

Na podstawie przygotowanej klasy utwórz egzemplarz `restaurant`. Wyświetl oddzielnie oba atrybuty, a następnie wywołaj obie metody.

**9.2. Trzy restauracje.** Pracę rozpoczęj od klasy opracowanej w ćwiczeniu 9.1. Utwórz trzy różne egzemplarze na podstawie tej klasy, a następnie wywołaj metodę `describe_restaurant()` dla każdego egzemplarza.

**9.3. Użytkownicy.** Przygotuj klasę o nazwie User. Zdefiniuj w niej dwa atrybuty (`first_name` i `last_name`), a następnie utwórz kilka innych atrybutów, które zwykle są przechowywane w profilu użytkownika. Zdefiniuj metodę o nazwie `describe_user()`, wyświetlającą podsumowanie informacji zebranych o użytkowniku. Utwórz jeszcze drugą metodę o nazwie `greet_user()`, która będzie wyświetlała użytkownikowi spersonalizowane powitanie.

Utwórz kilka egzemplarzy reprezentujących różnych użytkowników, a następnie dla każdego z nich wywołaj obie metody.

# Praca z klasami i egzemplarzami

Klasy można wykorzystać do przedstawienia wielu rzeczywistych sytuacji. Kiedy już zdefiniujesz klasę, większość czasu będziesz poświęcać na pracę z egzemplarzami utworzonymi na jej podstawie. Jednym z pierwszych zadań będzie modyfikacja atrybutów związanych z danym egzemplarzem klasy. Tę modyfikację atrybutów można przeprowadzić bezpośrednio w egzemplarzu, lub też przygotować metody uaktualniające atrybuty w określony sposób.

## Klasa Car

Zdefiniujemy teraz nową klasę, tym razem reprezentującą samochód. Nasza klasa będzie przechowywać informacje o samochodzie oraz zawierała metodę pozwalającą na wyświetlenie podsumowania tych informacji.

Plik `car.py`:

```
class Car:  
    """Prosta próba zaprezentowania samochodu."""  
  
    def __init__(self, make, model, year): ❶  
        """Inicjalizacja atrybutów opisujących samochód."""
```

```
self.make = make
self.model = model
self.year = year

def get_descriptive_name(self): ❷
    """Zwrot elegancko sformatowanego opisu samochodu."""
    long_name = f'{self.year} {self.make} {self.model}'
    return long_name.title()

my_new_car = Car('audi', 'a4', 2024) ❸
print(my_new_car.get_descriptive_name())
```

---

W wierszu ❶ klasy `Car` mamy zdefiniowaną metodę `__init__()` wraz z parametrem `self` umieszczonym jako pierwszy na liście, czyli podobnie jak w przypadku poprzednio omawianej klasy `Dog`. Ponadto lista parametrów zawiera jeszcze trzy inne pozycje: `make`, `model` i `year`. Metoda `__init__()` pobiera te parametry, a następnie przechowuje je w atrybutach, które będą powiązane z egzemplarzami utworzonymi na podstawie tej klasy. Podczas tworzenia nowego egzemplarza klasy `Car` musimy więc podać producenta, model i rok produkcji samochodu.

W wierszu ❷ mamy zdefiniowaną metodę o nazwie `get_descriptive_name()`, odpowiedzialną za umieszczenie w jednym ciągu tekstowym informacji pobranych z atrybutów `make`, `model` i `year` oraz wyświetlenie ich w formie elegancko sformatowanego opisu samochodu. Dzięki tej metodzie unikamy konieczności oddzielnego wyświetlania wartości poszczególnych atrybutów. Aby w tej metodzie móc pracować z wartościami atrybutów, wykorzystujemy notację `self.make`, `self.model` i `self.year`. Później w wierszu ❸ tworzymy egzemplarz na podstawie klasy `Car` i umieszczamy go w zmiennej `my_new_car`. Następnie wywołujemy metodę `get_descriptive_name()`, aby wyświetlić opis samochodu:

---

2024 Audi A4

---

Klasa stanie się jeszcze bardziej interesująca, kiedy zdefiniujemy w niej atrybut, którego wartość będzie się zmieniać w czasie. Dodajemy więc atrybut przechowujący aktualny przebieg samochodu.

## Przypisanie atrybutowi wartości domyślnej

Podczas tworzenia egzemplarza atrybuty mogą być definiowane bez konieczności przekazywania ich w postaci parametrów. Te atrybuty można zdefiniować w metodzie `__init__()`, w której zostaną im przypisane wartości domyślne.

Przechodzimy więc do dodania atrybutu o nazwie `odometer_reading`, który zawsze będzie miał wartość początkową wynoszącą 0. Ponadto zdefiniujemy metodę `read_odometer()` pomagającą robić odczyty licznika przebiegu w danym pojeździe:

---

```
class Car:

    def __init__(self, make, model, year):
        """Inicjalizacja atrybutów opisujących samochód."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0 1

    def get_descriptive_name(self):
        --cięcie--

    def read_odometer(self): 2
        """Wyświetla informację o przebiegu samochodu."""
        print(f"Ten samochód ma przejechane {self.odometer_reading} km.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

---

Kiedy Python wywoła metodę `__init__()` w celu utworzenia nowego egzemplarza, egzemplarz ten nadal będzie przechowywać producenta, model i rok produkcji samochodu jako atrybuty, podobnie jak to było w poprzednim przykładzie. Jednak tym razem utworzony zostanie również nowy atrybut o nazwie `odometer_reading`, którego wartością początkową będzie 0 (patrz wiersz **1**). Ponadto w wierszu **2** mamy nową metodę o nazwie `read_odometer()`, ułatwiającą odczytanie przebiegu danego pojazdu.

Na razie przebieg samochodu wynosi 0 km:

---

```
2024 Audi A4
Ten samochód ma przejechane 0 km.
```

---

Nie wszystkie samochody są sprzedawane z przebiegiem wynoszącym dokładnie 0 km na liczniku, więc potrzebny jest nam sposób pozwalający na zmianę wartości tego atrybutu.

## Modyfikacja wartości atrybutu

Wartość atrybutu można zmienić na trzy sposoby: zmienić wartość bezpośrednio w egzemplarzu, ustawić wartość za pomocą metody, lub też inkrementować wartość (czyli dodać do niej pewną wartość) za pomocą metody. Przeanalizujemy te raz wszystkie wymienione podejścia.

### Bezpośrednia modyfikacja wartości atrybutu

Najprostszy sposób na zmodyfikowanie wartości atrybutu polega na uzyskaniu dostępu do tego atrybutu bezpośrednio w egzemplarzu. Poniżej pokazałem, jak

za pomocą podejścia bezpośredniego można ustawić wartość licznika przebiegu na dokładnie 23 km:

---

```
class Car:  
    --cięcie--  
  
    my_new_car = Car('audi', 'a4', 2024)  
    print(my_new_car.get_descriptive_name())  
  
    my_new_car.odometer_reading = 23  
    my_new_car.read_odometer()
```

---

Użyliśmy notacji z kropką w celu uzyskania dostępu do atrybutu `odometer_reading` samochodu i bezpośredniego ustawienia jego wartości. Ten wiersz nakazuje Pythonowi pobrać egzemplarz `my_new_car`, odszukać powiązany z nim atrybut `odometer_reading` i przypisać temu atrybutowi wartości 23:

---

```
2024 Audi A4  
Ten samochód ma przejechane 23 km.
```

---

Czasami zachodzi potrzeba bezpośredniego uzyskania dostępu do atrybutu w pokazany powyżej sposób. Z kolei w innych sytuacjach oczekiwane jest zdefiniowanie metody odpowiedzialnej za uaktualnianie wartości atrybutu.

## Modyfikacja wartości atrybutu za pomocą metody

Użyteczne może być przygotowanie metod uaktualniających wartości określonych atrybutów. Zamiast bezpośrednio uzyskiwać dostęp do atrybutu, nową wartość po prostu przekazujesz metodzie, która wewnętrznie zajmuje się uaktualnieniem wartości atrybutu.

Poniżej przedstawiłem przykład metody o nazwie `update_odometer()`:

---

```
class Car:  
    --cięcie--  
  
    def update_odometer(self, mileage):  
        """Przypisanie podanej wartości licznikowi przebiegu samochodu."""  
        self.odometer_reading = mileage  
  
    my_new_car = Car('audi', 'a4', 2024)  
    print(my_new_car.get_descriptive_name())  
  
    my_new_car.update_odometer(23) ❶  
    my_new_car.read_odometer()
```

---

Jedyna zmiana w klasie `Car` polega na dodaniu metody `update_odometer()`. Ta metoda pobiera bieżący przebieg pojazdu i umieszcza tę wartość w atrybutie `self.odometer_reading`. W wierszu ❶ widzimy wywołanie metody `update_odometer()` i podanie liczby 23 jako argumentu (odpowiada on parametrowi `mileage` w definicji metody). Działanie metody polega na przypisaniu licznikowi przebiegu samochodu podanej wartości 23, co potwierdzają dane wyjściowe wyświetcone przez metodę `read_odometer()`:

---

```
2024 Audi A4
Ten samochód ma przejechane 23 km.
```

---

Metodę `update_odometer()` można jeszcze bardziej rozbudować, aby wykonywała też inne zadania w trakcie każdego odczytu licznika przebiegu. Przykładowo zaimplementujemy teraz logikę uniemożliwiającą cofnięcie licznika przebiegu:

---

```
class Car:
    --cięcie--

    def update_odometer(self, mileage):
        """
        Przypisanie podanej wartości licznikowi przebiegu samochodu.
        Zmiana zostanie odrzucona w przypadku próby cofnięcia licznika.
        """
        if mileage >= self.odometer_reading: ❶
            self.odometer_reading = mileage
        else:
            print("Nie można cofnąć licznika przebiegu samochodu!") ❷
```

---

Teraz przed faktycznym wprowadzeniem zmiany metoda `update_odometer()` sprawdza, czy nowa wartość ma sens. Kiedy nowa wartość przebiegu (`mileage`) jest wyższa niż lub równa dotychczasowej (`self.odometer_reading`), wówczas można ją uaktualnić (patrz wiersz ❶). Natomiast w przypadku gdy nowa wartość przebiegu jest mniejsza od bieżącej, użytkownikowi zostaje wyświetlona informacja, że nie wolno cofać licznika przebiegu (patrz wiersz ❷).

## Inkrementacja wartości atrybutu za pomocą metody

Czasami zachodzi potrzeba inkrementowania wartości atrybutu o określona wartość zamiast przypisywania mu zupełnie nowej wartości. Przyjmujemy założenie, że użytkownik kupił używany samochód i przejechał nim 100 km od chwili zakupu do momentu zarejestrowania pojazdu. Poniżej przedstawiłem metodę pozwalającą na przekazanie wartości inkrementacji oraz jej dodanie do bieżącej wartości licznika przebiegu samochodu:

---

```
class Car:  
    --cięcie--  
  
    def update_odometer(self, mileage):  
        --cięcie--  
  
    def increment_odometer(self, kilometers):  
        """Inkrementacja wartości licznika przebiegu samochodu o podaną wartość."""  
        self.odometer_reading += kilometers  
  
my_used_car = Car('subaru', 'outback', 2019) ❶  
print(my_used_car.get_descriptive_name())  
  
my_used_car.update_odometer(23_500) ❷  
my_used_car.read_odometer()  
  
my_used_car.increment_odometer(100)  
my_used_car.read_odometer()
```

---

Nowa metoda `increment_odometer()` pobiera aktualny przebieg, a następnie dodaje tę wartość do aktualnie przechowywanej w atrybucie `self.odometer_reading`. W wierszu ❶ tworzymy nowy egzemplarz o nazwie `my_used_car`, reprezentujący używany samochód. Przebieg tego pojazdu określamy na 23 500 km przez wywołanie metody `update_odometer()` wraz z argumentem `23_500` (patrz wiersz ❷). Na koniec wywołujemy metodę `increment_odometer()` wraz z argumentem `100`, co powoduje dodanie tej wartości do aktualnego przebiegu samochodu:

---

```
2019 Subaru Outback  
Ten samochód ma przejechane 23500 km.  
Ten samochód ma przejechane 23600 km.
```

---

Omówioną tutaj metodę można bardzo łatwo zmodyfikować, aby odrzuciła ujemną wartość inkrementacji, co uniemożliwi użytkownikowi cofnięcie licznika przebiegu samochodu.

**UWAGA** *Tego rodzaju metody można wykorzystywać do kontrolowania sposobu, w jaki użytkownicy programu uaktualniają wartości takie jak wartość licznika przebiegu samochodu. Jednak każdy, kto będzie miał dostęp do programu, będzie mógł ustawić dowolną wartość licznika przebiegu, wykorzystując do tego bezpośredni dostęp do odpowiedniego atrybutu. Skuteczne zabezpieczenia wymagają wyjątkowej ostrożności i zwracania uwagi na szczegóły. Nie można się ograniczyć jedynie do prostego sprawdzenia, takiego jak w omówionym powyżej przykładzie.*

## ZRÓB TO SAM

**9.4. Liczba obsłużonych.** Pracę rozpocznij od programu utworzonego w ćwiczeniu 9.1. Dodaj atrybut o nazwie `number_served` o wartości domyślnej 0. Następnie na podstawie klasy utwórz egzemplarz o nazwie `restaurant`. Wyświetl liczbę klientów obsłużonych przez restaurację, zmień tę wartość i później wyświetl nową.

Dodaj metodę o nazwie `set_number_served()`, pozwalającą na zdefiniowanie liczby obsłużonych klientów. Wywołaj tę metodę wraz z różnymi wartościami i je wyświetl.

Dodaj metodę o nazwie `increment_number_served()`, pozwalającą na inkrementację wartości wskazującej na liczbę obsłużonych klientów. Wywołaj tę metodę dowolną ilość razy, symulując w ten sposób liczbę klientów obsłużonych na przykład w ciągu dnia roboczego.

**9.5. Próby logowania.** Pracę rozpocznij od programu utworzonego w ćwiczeniu 9.3. Do klasy `User` dodaj metodę o nazwie `increment_login_attempts()`, pozwalający na inkrementację wartości `login_attempts` o 1. Utwórz drugą metodę o nazwie `reset_login_attempts()`, która będzie zerowała wartość `login_attempts`.

Utwórz egzemplarz klasy `User` i kilkukrotnie wywołaj metodę `increment_login_attempts()`. Wyświetl wartość `login_attempts`, aby mieć pewność o jej prawidłowej inkrementacji. Następnie wywołaj metodę `reset_login_attempts()`. Ponownie wyświetl wartość `login_attempts` i sprawdź, czy na pewno wynosi 0.

# Dziedziczenie

Kiedy przygotowuje się klasę, pracę nie zawsze trzeba zaczynać zupełnie od początku. Jeżeli tworzona klasa ma być wyspecjalizowaną wersją innej opracowanej wcześniej klasy, można skorzystać z *dziedziczenia*. Gdy jedna klasa *dziedzicy* po innej, automatycznie pobiera wszystkie atrybuty i metody z tej klasy. Klasa pierwotna jest nazywana *klasą nadzczną*, natomiast nowa jest określana mianem *klasy potomnej*. Ta klasa potomna dziedziczy wszystkie atrybuty i metody po klasie nadzędnej, choć może definiować również nowe atrybuty i metody.

## Metoda `__init__()` w klasie potomnej

Podezas tworzenia egzemplarza na podstawie klasy potomnej bardzo często będziesz wywoływać metodę `__init__()` klasy nadzędnej. To spowoduje inicjalizację wszystkich atrybutów zdefiniowanych w klasie nadzędnej i udostępnienie ich w klasie potomnej.

Rozważmy na przykład samochód napędzany silnikiem elektrycznym. Tego rodzaju pojazd to specjalny typ samochodu, więc naszą nową klasę `ElectricCar`

możemy zbudować w oparciu o utworzoną wcześniej klasę `Car`. W ten sposób konieczne będzie przygotowanie jedynie kodu dla atrybutów i metod dotyczących samochodów napędzanych silnikami elektrycznymi.

Pracę rozpoczętymy od utworzenia prostej wersji klasy `ElectricCar`, która będzie miała dokładnie takie same możliwości jak przygotowana wcześniej klasa `Car`.

#### Plik `electric_car.py`:

---

```
class Car: ❶
    """Prosta próba zaprezentowania samochodu."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Zwrot elegancko sformatowanego opisu samochodu."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Wyświetlenie informacji o przebiegu samochodu."""
        print(f"Ten samochód ma przejechane {self.odometer_reading} km.")

    def update_odometer(self, mileage):
        """
        Przypisanie podanej wartości licznikowi przebiegu samochodu.
        Zmiana zostanie odrzucona w przypadku próby cofnięcia licznika.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Nie można cofnąć licznika przebiegu samochodu!")

    def increment_odometer(self, kilometers):
        """
        Inkrementacja wartości licznika przebiegu samochodu o podaną
        wartość."""
        self.odometer_reading += kilometers

class ElectricCar(Car): ❷
    """Przedstawia cechy charakterystyczne samochodu elektrycznego."""

    def __init__(self, make, model, year): ❸
        """Inicjalizacja atrybutów klasy nadzędnej."""
        super().__init__(make, model, year) ❹

    my_leaf = ElectricCar('nissan', 'leaf', 2024) ❺
    print(my_leaf.get_descriptive_name())
```

---

W wierszu ① pracę rozpoczynamy od istniejącej już klasy `Car`. Zanim utworzymy klasę potomną, klasa nadziedzicząca musi być częścią bieżącego pliku oraz musi się już w nim znajdować. Następnie w wierszu ② mamy definicję klasy potomnej o nazwie `ElectricCar`. Nazwa klasy nadziedziczącej musi być umieszczona w nawiasie definicji klasy potomnej. Metoda `__init__()` w klasie potomnej (patrz wiersz ③) pobiera informacje wymagane do utworzenia egzemplarza klasy `Car`.

Wywołanie `super()` w wierszu ④ to funkcja specjalna pomagająca Pythonowi w utworzeniu połączenia między klasami nadziedziczącą i potomną. Ten wiersz nakazuje wywołanie metody `__init__()` z klasy nadziedziczącej dla `ElectricCar`, co pozwala egzemplarzowi klasy `ElectricCar` otrzymać wszystkie atrybuty jego klasy nadziedziczącej. Nazwa funkcji `super` pochodzi od konwencji określania klasy nadziedziczącej mianem *superklasy*, a klasy potomnej mianem *podklasti*.

Możemy teraz sprawdzić, czy dziedziczenie działa prawidłowo. W tym celu spróbujemy utworzyć egzemplarz samochodu elektrycznego, wykorzystując do tego te same informacje, które wcześniej podaliśmy podczas tworzenia egzemplarza tradycyjnego samochodu. W wierszu ⑤ tworzymy więc egzemplarz klasy `ElectricCar` i umieszczać go w zmiennej `my_leaf`. Ten wiersz zawiera wywołanie metody `__init__()` zdefiniowanej w klasie `ElectricCar`, która z kolei nakazuje Pythonowi wywołanie metody `__init__()` zdefiniowanej w klasie nadziedziczącej, czyli `Car`. W wywołaniu przekazujemy argumenty `'nissan'`, `'leaf'` i `2024`.

Oprócz metody `__init__()` nie istnieją jeszcze żadne atrybuty lub metody charakterystyczne dla samochodu z napędem elektrycznym. Na tym etapie po prostu tworzymy egzemplarz dla samochodu elektrycznego, który będzie zachowywał się dokładnie tak samo jak samochód reprezentowany przez egzemplarz klasy `Car`:

---

2024 Nissan Leaf

---

Przekonaliśmy się, że egzemplarz klasy `ElectricCar` działa tak samo jak egzemplarz klasy `Car`, możemy więc przystąpić do definiowania atrybutów i metod charakterystycznych dla samochodów z napędem elektrycznym.

## Definiowanie atrybutów i metod dla klasy potomnej

Po przygotowaniu klasy potomnej dziedziczącej po klasie nadziedziczącej możemy przystąpić do dodawania nowych atrybutów i metod, które są niezbędne, aby móc odróżnić tę klasę potomną od jej klasy nadziedziczącej.

Teraz dodamy więc atrybut charakterystyczny dla samochodu napędzanego silnikiem elektrycznym, na przykład akumulatory, oraz metodę wyświetlającą informacje o tym atrybutu. W nowym atrybutu będziemy przechowywać wielkość akumulatora oraz przygotujemy metodę wyświetlającą opis tego akumulatora:

---

```
class Car:  
    --cięcie--  
  
    class ElectricCar(Car):  
        """Przedstawia cechy charakterystyczne samochodu elektrycznego."""  
  
        def __init__(self, make, model, year):  
            """  
            Inicjalizacja atrybutów klasy nadzędnej.  
            Następnie inicjalizacja atrybutów charakterystycznych  
            dla samochodu elektrycznego.  
            """  
            super().__init__(make, model, year)  
            self.battery_size = 40 ❶  
  
        def describe_battery(self): ❷  
            """Wyświetlenie informacji o wielkości akumulatora."""  
            print(f"Ten samochód ma akumulator o pojemności {self.battery_size} kWh.")  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())  
my_leaf.describe_battery()
```

---

W wierszu ❶ dodajemy nowy atrybut o nazwie `self.battery_size` i przypisujemy mu wartość początkową wynoszącą na przykład 40. Ten atrybut będzie powiązany ze wszystkimi egzemplarzami utworzonymi na podstawie klasy `ElectricCar`, ale nie zostanie uwzględniony w żadnym egzemplarzu klasy `Car`. Ponadto dodajemy metodę o nazwie `describe_battery()` wyświetlającą informacje o akumulatorze (patrz wiersz ❷). Kiedy wywołamy tę metodę, na ekranie zostanie wyświetlony komunikat jasno wskazujący, że mamy do czynienia z samochodem o napędzie elektrycznym:

---

```
2024 Nissan Leaf  
Ten samochód ma akumulator o pojemności 40 kWh.
```

---

Nie ma żadnych ograniczeń w zakresie specjalizacji klasy `ElectricCar`. Możesz dodać dowolną liczbę atrybutów i metod niezbędnych do modelowania samochodu o napędzie elektrycznym na zadowalającym Cię poziomie dokładności. Te atrybuty lub metody, które można zastosować względem dowolnego samochodu, a nie tylko względem samochodów z napędem elektrycznym, powinny być dodawane do klasy `Car` zamiast do `ElectricCar`. W ten sposób każdy użytkownik klasy `Car` również będzie miał dostęp do tych funkcjonalności, natomiast użytkownicy klasy `ElectricCar` będą mieli do dyspozycji kod obsługujący informacje i zachowania typowe jedynie dla samochodów elektrycznych.

## Nadpisywanie metod klasy nadzędnej

Istnieje możliwość nadpisania dowolnej metody klasy nadzędnej, która nie będzie pasowała do modelu tworzonego za pomocą klasy potomnej. W tym celu w klasie potomnej należy zdefiniować metodę o takiej samej nazwie jak nadpisywana metoda klasy nadzędnej. Python nie będzie zwracać uwagi na tę metodę w klasie nadzędnej i uwzględnii jedynie metodę zdefiniowaną w klasie potomnej.

Przyjmujemy założenie, że klasa `Car` zawiera metodę o nazwie `fill_gas_tank()` odpowiedzialną za obsługę tankowania samochodu. Ponieważ ta metoda jest bezcelowa w samochodzie o napędzie elektrycznym, możemy ją nadpisać w klasie potomnej. Oto jedno z możliwych rozwiązań.

---

```
def ElectricCar(Car):
    --cięcie--

    def fill_gas_tank():
        """Samochód o napędzie elektrycznym nie ma zbiornika paliwa."""
        print("Ten samochód nie wymaga tankowania paliwa!")
```

---

Jeżeli teraz ktokolwiek spróbuje wywołać metodę `fill_gas_tank()` dla egzemplarza reprezentującego samochód elektryczny, Python zignoruje tę metodę w klasie `Car` i wykoná wersję zdefiniowaną w klasie potomnej. Kiedy stosujesz dziedziczenie, wówczas upewnij się, że klasy potomne zawierają jedynie niezbędne im atrybuty i metody oraz nadpisują wszystko to, co z klasy nadzędnej jest im niepotrzebne.

## Egzemplarz jako atrybut

Kiedy będziesz modelować w kodzie coś pochodzące ze świata rzeczywistego, może się okazać, że będziesz dodawać coraz więcej szczegółów do klasy. Zdasz sobie sprawę z rosnącej listy atrybutów i metod oraz ze zwiększającego się rozmiaru plików. W takiej sytuacji będziesz mógł rozważyć przygotowanie pewnej części klasy jako zupełnie oddzielnej klasy. W ten sposób ogromną klasę podzieliś na kilka mniejszych, które będą ze sobą współdziałać. Takie podejście jest nazywane *kompozycją*.

Jeżeli na przykład będziemy kontynuować dodawanie kolejnych szczegółów do klasy `ElectricCar`, może się okazać, że dodaliśmy wiele atrybutów i metod dotyczących akumulatora samochodu napędzanego silnikiem elektrycznym. W takim przypadku dobrym rozwiązaniem będzie przeniesienie tych wszystkich atrybutów i metod do oddzielnej klasy o nazwie `Battery`. Następnie egzemplarza klasy `Battery` będziemy mogli użyć jako atrybutu w klasie `ElectricCar`:

---

```
class Car:
    --cięcie--

class Battery:
    """Prosta próba modelowania akumulatora samochodu elektrycznego."""
```

---

```
def __init__(self, battery_size=40): ❶
    """Inicjalizacja atrybutów akumulatora."""
    self.battery_size = battery_size

def describe_battery(self): ❷
    """Wyświetlenie informacji o wielkości akumulatora."""
    print(f"Ten samochód ma akumulator o pojemności {self.battery_size} kWh.")

class ElectricCar(Car):
    """Predstawia cechy charakterystyczne samochodu elektrycznego."""

    def __init__(self, make, model, year):
        """
        Inicjalizacja atrybutów klasy nadzędnej.
        Następnie inicjalizacja atrybutów charakterystycznych
        dla samochodu elektrycznego.
        """
        super().__init__(make, model, year)
        self.battery = Battery() ❸

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
```

---

Definiujemy nową klasę o nazwie `Battery`, która nie dziedziczy po żadnej innej klasie. Przedstawiona w wierszu ❶ metoda `__init__()` ma tylko jeden — oczywiście pomijając domyślny `self` — parametr o nazwie `battery_size`. Jest to parametr opcjonalny, przeznaczony do zdefiniowania pojemności akumulatora. Jeżeli jego wartość nie zostanie wyraźnie podana, domyślnie będzie wynosić 40. Metoda `describe_battery()` również została przeniesiona do nowej klasy (patrz wiersz ❷).

W klasie `ElectricCar` mamy teraz atrybut o nazwie `self.battery` (patrz wiersz ❸). Ten wiersz nakazuje Pythonowi utworzenie nowego egzemplarza klasy `Battery` (wraz z wartością domyślną 40 dla parametru `battery_size`, ponieważ nie podaliśmy jego wartości) i umieszczenie go w atrybucie `self.battery`. To będzie wykonywane w trakcie każdego wywołania metody `__init__()` i tym samym wszystkie egzemplarze klasy `ElectricCar` od tej chwili będą miały automatycznie tworzony egzemplarz klasy `Battery`.

Tworzymy egzemplarz samochodu o napędzie elektrycznym i przechowujemy go w zmiennej `my_leaf`. Kiedy zajdzie potrzeba opisania akumulatora, zrobimy to za pomocą atrybutu `battery`:

---

```
my_leaf.battery.describe_battery()
```

---

Powyższy wiersz nakazuje Pythonowi odszukanie egzemplarza `my_leaf`, znalezienie jego atrybutu `battery` i wywołanie metody o nazwie `describe_battery()`, powiązanej z egzemplarzem klasy `Battery` przechowywanym w atrybucie.

Wygenerowane dane wyjściowe są dokładnie takie same jak w poprzednim przykładzie:

---

```
2024 Nissan Leaf  
Ten samochód ma akumulator o pojemności 40 kWh.
```

---

Można odnieść wrażenie, że wykonaliśmy dużą ilość dodatkowej pracy. Jednak zyskaliśmy możliwość szczegółowego opisania akumulatora bez zaśmiecania przy tym klasy `ElectricCar`. Dodamy teraz do klasy `Battery` następną metodę, odpowiedzialną za wyświetlenie zasięgu samochodu na podstawie pojemności użytego w nim akumulatora:

---

```
class Car:  
    --cięcie--  
  
class Battery:  
    --cięcie--  
  
    def get_range(self):  
        """  
        Wyświetla informacje o zasięgu samochodu na podstawie pojemności  
        akumulatora.  
        """  
        if self.battery_size == 40:  
            range = 150  
        elif self.battery_size == 65:  
            range = 225  
  
        print(f"Zasięg tego samochodu wynosi około {range} km po pełnym  
        naładowaniu akumulatora.")  
  
class ElectricCar(Car):  
    --cięcie--  
  
    my_leaf = ElectricCar('nissan', 'leaf', 2024)  
    print(my_leaf.get_descriptive_name())  
    my_leaf.battery.describe_battery()  
    my_leaf.battery.get_range() ❶
```

---

Nowa metoda `get_range()` przeprowadza pewną prostą analizę. W przypadku akumulatora o pojemności 40 kWh metoda podaje zasięg samochodu wynoszący 150 km, natomiast w przypadku akumulatora o pojemności 65 kWh zasięg wynoszący 225 km. Następnie ta wartość zostaje wyświetlona. Kiedy będziemy chcieli użyć tej metody, będziemy musieli ją wywołać za pomocą atrybutu `battery`, tak jak pokazalem w wierszu ❶.

Wygenerowane dane wyjściowe podają zasięg samochodu ustalony na podstawie pojemności akumulatora:

---

2024 Nissan Leaf

Ten samochód ma akumulator o pojemności 40 kWh.

Zasięg tego samochodu wynosi około 150 km po pełnym naładowaniu akumulatora.

---

## Modelowanie rzeczywistych obiektów

Kiedy rozpocznesz modelowanie znacznie bardziej skomplikowanych rzeczy, takich jak samochód o napędzie elektrycznym, wówczas będziesz zmagać się z interesującymi problemami. Na przykład z takim, czy zasięg samochodu elektrycznego jest właściwością akumulatora czy samochodu. Jeżeli opisujemy tylko jeden samochód, prawdopodobnie nie będzie błędem powiązanie metody `get_range()` z klasą `Battery`. Jednak w przypadku opisywania całej linii samochodów danego producenta metodę `get_range()` prawdopodobnie trzeba będzie przenieść do klasy `ElectricCar`. Wprawdzie ta metoda będzie sprawdzać pojemność akumulatora przed wyświetleniem informacji o zasięgu samochodu, ale podawane przez nią dane będą dotyczyć konkretnego pojazdu. Alternatywne podejście polega na zachowaniu powiązania między metodą `get_range()` i akumulatorem oraz na przekazywaniu danych za pomocą parametru takiego jak `car_model`. W takim przypadku metoda `get_range()` przedstawi informacje o zasięgu samochodu na podstawie pojemności akumulatora i modelu samochodu.

W ten sposób dochodzimy do interesującego momentu na Twojej ścieżce rozwoju jako programisty. Kiedy zaczniesz szukać odpowiedzi na tego rodzaju pytania, będzie to oznaczało, że zacząłeś myśleć na wyższym poziomie logicznym i nie koncentrujesz się jedynie na składni. Myślisz nie tylko o Pythonie, lecz także zastanawiasz się, jak przedstawić rzeczywistość w kodzie. Gdy dotrzesz do tego punktu, przekonasz się, że często nie można jednoznacznie określić, czy dane podejście do modelowania rzeczywistej sytuacji jest właściwe lub niewłaściwe. Niektóre podejścia są znacznie efektywniejsze niż inne, ale odszukanie najbardziej efektywnego wymaga nieco praktyki. Jeżeli tworzony przez Ciebie kod działa zgodnie z oczekiwaniemi, to znaczy, że dobrze wykonujesz swoją pracę! Nie zniechęcaj się, gdy się okaże, że wielokrotnie modyfikujesz klasy i stosujesz w nich różne podejścia. W poszukiwaniu rozwiązania, które pozwali tworzyć odpowiedni i efektywnie działający kod, każdy programista przechodzi przez ten etap.

### ZRÓB TO SAM

**9.6. Budka z lodami.** Budka z lodami to szczególny rodzaj restauracji. Zdefiniuj klasę o nazwie `IceCreamStand` dziedziczącą po klasie `Restaurant` utworzonej w ćwiczeniu 9.1 lub 9.4. Dowolna wersja wymienionej klasy będzie się sprawdzać — po prostu wybierz jedną z nich. Dodaj atrybut o nazwie `flavors`, przechowujący listę różnych smaków lodów. Zdefiniuj metodę wyświetlającą dostępne smaki lodów. Utwórz egzemplarz klasy `IceCreamStand` i wywołaj nową metodę.

**9.7. Admin.** Administrator to specjalny rodzaj użytkownika. Zdefiniuj klasę o nazwie `Admin` dziedziczącą po klasie `User` utworzonej w ćwiczeniu 9.3 lub 9.5. Dodaj atrybut `privileges` przechowujący listę ciągów tekstowych takich jak „może dodać post”, „może usunąć post” czy „może zbanować użytkownika”. Zdefiniuj metodę o nazwie `show_privileges()`, wyświetlającą listę uprawnień administratora. Utwórz egzemplarz klasy `Admin` i wywołaj nową metodę.

**9.8. Uprawnienia.** Zdefiniuj oddzielną klasę `Privileges`. Ta klasa powinna mieć jeden atrybut (`privileges`), przechowujący listę ciągów tekstowych, tak jak przedstawiłem to w poprzednim ćwiczeniu. Metodę `show_privileges()` przenieś do nowej klasy. Utwórz egzemplarz klasy `Privileges` jako atrybut w klasie `Admin`. Następnie utwórz nowy egzemplarz klasy `Admin` i użyj metody `show_privileges()` do wyświetlenia uprawnień.

**9.9. Aktualnienie akumulatora.** Pracę rozpocznij od ostatniej wersji programu `electric_car.py` utworzonej w tym podrozdziale. Do klasy `Battery` dodaj nową metodę o nazwie `upgrade_battery()`. Ta metoda powinna sprawdzić pojemność akumulatora i ustawić ją na 65, jeśli aktualnie jest inna. Utwórz egzemplarz samochodu o napędzie elektrycznym wraz z akumulatorem o domyślnej pojemności, wywołaj metodę `get_range()`, a następnie metodę `upgrade_battery()` i ponownie `get_range()`. Powinieneś zauważyć zwiększenie zasięgu samochodu.

## Import klas

Wraz z dodawaniem kolejnych funkcjonalności do klas pliki będą stawały się coraz większe, nawet w przypadku prawidłowo stosowanego dziedziczenia. Aby pozostać w zgodzie z ogólną filozofią Pythona, w plikach powinniśmy umieszczać kod źródłowy pozbawiony niepotrzebnych elementów. Python pomaga w osiągnięciu tego celu, umożliwiając przechowywanie klas w modułach. Poszczególne klasy można importować do programu głównego wtedy, gdy będą potrzebne.

### Import pojedynczej klasy

Rozpoczynamy od utworzenia modułu zawierającego jedynie klasę `Car`. Od razu pojawia się problem związany z nazwą, ponieważ w tym rozdziale już utworzyliśmy plik o nazwie `car.py`. Jednak to moduł powinien mieć tę nazwę, skoro zawiera kod przedstawiający samochód. Rozwiążemy więc problem w ten sposób, że umieścimy klasę `Car` w module `car.py`, zastępując tym samym wcześniej utworzony plik o tej samej nazwie. Od tej chwili każdy program używający naszego modułu musi mieć inną nazwę pliku, na przykład `my_car.py`. Poniżej przedstawitem zawartość pliku `car.py`, na którą składa się po prostu kod klasy `Car`.

### Plik car.py:

---

```
"""Klasa, która będzie używana do zaprezentowania samochodu.""" ①
class Car:

    """Prosta próba zaprezentowania samochodu."""

    def __init__(self, make, model, year):
        """Inicjalizacja atrybutów opisujących samochód."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Zwrot elegancko sformatowanego opisu samochodu."""
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    def read_odometer(self):
        """Wyświetla informację o przebiegu samochodu."""
        print(f'Ten samochód ma przejechane {self.odometer_reading} km.')

    def update_odometer(self, mileage):
        """
        Przypisanie podanej wartości licznikowi przebiegu samochodu.
        Zmiana zostanie odrzucona w przypadku próby cofnięcia licznika.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("Nie można cofnąć licznika przebiegu samochodu!")

    def increment_odometer(self, kilometers):
        """Inkrementacja wartości licznika przebiegu samochodu o podaną wartość."""
        self.odometer_reading += kilometers
```

---

W wierszu ① umieściłem komentarz typu *docstring* o zasięgu modułu, pokrótce opisujący przeznaczenie zawartości tego modułu. Komentarz typu *docstring* należy umieszczać na początku każdego przygotowywanego modułu.

Następnym krokiem jest utworzenie nowego pliku o nazwie *my\_car.py*. W tym pliku zaimportujemy klasę Car i później utworzymy egzemplarz na jej podstawie.

### Plik my\_car.py:

---

```
from car import Car ①

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
```

```
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

---

Polecenie `import` w wierszu ❶ nakazuje Pythonowi otworzenie modułu `car.py` i zainportowanie klasy `Car`. W tym momencie klasę `Car` możemy używać, jakby została zdefiniowana w bieżącym pliku programu. Wygenerowane dane wyjściowe są dokładnie takie same jak wcześniej:

---

```
2024 Audi A4
Ten samochód ma przejechane 23 km.
```

---

Importowanie klas zalicza się do efektywnych technik programowania. Zastanów się, jak długi byłby nasz program, gdy zawierał kod całej klasy `Car`. Kiedy przeniesiemy klasę do modułu, a następnie go zainportujemy, nadal zachowamy tę samą funkcjonalność, a program główny pozostanie niezaśmiecony i łatwy do odczytania. Ponadto większość jego logiki będzie przechowywana w oddzielnych plikach. Kiedy zdefiniowane klasy będą działały zgodnie z oczekiwaniami, wówczas będziemy mogli zapomnieć o ich plikach i zamiast tego skoncentrować się na logice wyższego poziomu stosowanej w programie głównym.

## Przechowywanie wielu klas w module

W pojedynczym module można przechowywać dowolną liczbę klas, choć poszczególne klasy w module powinny być w jakiś sposób ze sobą powiązane. Ponieważ klasy `Battery` i `ElectricCar` pomagają w zaprezentowaniu samochodu, możemy je umieścić w module `car.py`.

*Plik car.py:*

---

```
"""
Zestaw klas przeznaczonych do zaprezentowania samochodu,
zarówno o napędzie tradycyjnym, jak i elektrycznym.
"""

class Car:
    """Cięcie--"""

class Battery:
    """Prosta próba modelowania akumulatora samochodu elektrycznego."""

    def __init__(self, battery_size=40):
        """Inicjalizacja atrybutów akumulatora."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Wyświetlenie informacji o wielkości akumulatora."""
        print(f"Ten samochód ma akumulator o pojemności {self.battery_size} kWh.")
```

```
def get_range(self):
    """Wyświetla informacje o zasięgu samochodu na podstawie pojemności
    akumulatora."""
    if self.battery_size == 40:
        range = 150
    elif self.battery_size == 65:
        range = 225

    print(f"Zasięg tego samochodu wynosi około {range} km po pełnym
    naładowaniu akumulatora.")

class ElectricCar(Car):
    """ Przedstawia cechy charakterystyczne samochodu elektrycznego."""

    def __init__(self, make, model, year):
        """
        Inicjalizacja atrybutów klasy nadrzędnej.
        Następnie inicjalizacja atrybutów charakterystycznych
        dla samochodu elektrycznego.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

---

Teraz możemy już utworzyć nowy plik o nazwie *my\_electric\_car.py*, zaimportować klasę ElectricCar i zdefiniować egzemplarz samochodu o napędzie elektrycznym.

*Plik my\_electric\_car.py:*

---

```
from car import ElectricCar

my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
my_leaf.battery.describe_battery()
my_leaf.battery.get_range()
```

---

Otrzymamy dokładnie te same dane wyjściowe jak we wcześniejszych przykładach, choć większość logiki została ukryta w module:

---

```
2024 Nissan Leaf
Ten samochód ma akumulator o pojemności 40 kWh.
Zasięg tego samochodu wynosi około 150 km po pełnym naładowaniu akumulatora.
```

---

## Import wielu klas z modułu

W programie można zaimportować dowolną liczbę klas. Gdy w tym samym pliku tworzymy egzemplarze samochodów o napędzie tradycyjnym i elektrycznym, wówczas konieczne będzie zainportowanie obu klas, czyli Car i ElectricCar.

### Plik my\_cars.py:

---

```
from car import Car, ElectricCar ①  
  
my_mustang = Car('ford', 'mustang', 2024) ②  
print(my_mustang.get_descriptive_name())  
my_leaf = ElectricCar('nissan', 'leaf', 2024) ③  
print(my_leaf.get_descriptive_name())
```

---

Jak pokazałem w wierszu ①, import wielu klas odbywa się przez podanie ich nazw rozdzielonych przecinkami. Po zimportowaniu niezbędnych klas można utworzyć dowolną liczbę egzemplarzy na podstawie każdej z klas.

W omawianym przykładzie utworzyliśmy egzemplarz reprezentujący tradycyjny samochód Ford Mustang (patrz wiersz ②) oraz samochód o napędzie elektrycznym Nissan Leaf (patrz wiersz ③):

---

```
2024 Ford Mustang  
2024 Nissan Leaf
```

---

## Import całego modułu

Istnieje możliwość zimportowania całego modułu, a następnie uzyskania dostępu do niezbędnych klas za pomocą notacji z kropką. Tego rodzaju rozwiązanie jest proste i powoduje, że kod jest łatwy do odczytania. Ponieważ każde wywołanie tworzy egzemplarz klasy zawierający nazwę modułu, unikamy konfliktów nazw z jakimkolwiek nazwami stosowanymi w bieżącym pliku.

Poniżej przedstawiłem wersję programu, w której importujemy cały moduł car, a następnie tworzymy dwa egzemplarze samochodów: pierwszy z napędem tradycyjnym i drugi z napędem elektrycznym.

### Plik my\_cars.py:

---

```
import car ①  
  
my_mustang = car.Car('ford', 'mustang', 2024) ②  
print(my_mustang.get_descriptive_name())  
  
my_leaf = car.ElectricCar('nissan', 'leaf', 2024) ③  
print(my_leaf.get_descriptive_name())
```

---

W wierszu ① importujemy cały moduł car. Następnie do niezbędnych nam klas dostęp uzyskujemy za pomocą składni *nazwa\_modułu.nazwa\_klasy*. Później tworzymy samochody: tradycyjny Ford Mustang (patrz wiersz ②) i elektryczny Nissan Leaf (patrz wiersz ③).

## Import wszystkich klas z modułu

Za pomocą poniższej składni można zaimportować wszystkie klasy zdefiniowane w module:

```
from module_name import *
```

Jednak powyższe podejście jest niezalecane z przynajmniej dwóch powodów. Po pierwsze, odczytując polecenie `import` na początku pliku, powinniśmy mieć możliwość ustalenia, jakie klasy są wykorzystywane przez dany program. W przypadku przedstawionego podejścia nie mamy jasnego wskazania, które klasy modułu są używane przez program. Po drugie, to podejście prowadzi do powstania niejasności związanych z nazwami w pliku. Jeżeli przypadkowo zaimportujesz klasę o takiej samej nazwie jak komponent istniejący w pliku programu, może dojść do powstania błędów trudnych do wykrycia. Wprawdzie nie zaleca się stosowania powyższej składni, ale zdecydowałem się na jej pokazanie, ponieważ prawdopodobnie spotkasz się z nią w kodzie tworzonym przez innych programistów.

Gdy zachodzi potrzeba zaimportowania wielu klas z modułu, wówczas lepszym rozwiązaniem jest import całego modułu i stosowanie składni *nazwa\_modułu*.  
→*nazwa\_klasy*. W przypadku takiego podejścia na początku pliku nie znajdziesz listy wszystkich klas użytych w bieżącym pliku, ale przynajmniej będą jasno wskazane moduły wykorzystane w programie. Ponadto unikniesz potencjalnych konfliktów nazw, które mogłyby wystąpić po zaimportowaniu wszystkich klas znajdujących się w module.

## Import modułu w module

Czasami chcesz rozmieścić klasy w wielu modułach, aby uniknąć zbyt dużego rozrośnięcia się jednego pliku oraz uniknąć przechowywania w tym samym module niepowiązanych ze sobą klas. W przypadku przechowywania klas w wielu modułach może się okazać, że działanie klasy w jednym module zależy od klasy znajdującej się w innym module. Kiedy będziesz mieć do czynienia z tego rodzaju sytuacją, najpierw zaimportuj klasę wymaganą przez klasę w innym module.

Przykładowo klasę `Car` będziemy przechowywać w jednym module, natomiast klasy `ElectricCar` i `Battery` w zupełnie oddzielnym. Przystępujemy więc do utworzenia nowego modułu o nazwie `electric_car.py` — zastępując utworzony wcześniej plik `electric_car.py` — i kopujemy do niego jedynie klasy `Battery` i `ElectricCar`.

*Plik electric\_car.py:*

```
"""Zestaw klas przeznaczonych do zaprezentowania samochodu elektrycznego."""

from car import Car
```

```
class Battery:  
    --cięcie--  
  
class ElectricCar(Car):  
    --cięcie--
```

---

Klasa `ElectricCar` wymaga uzyskania dostępu do jej klasy nadzędnej (`Car`), więc bezpośrednio importujemy klasę `Car` w tym module. Jeżeli zapomnisz o tym wierszu, Python zgłosi błąd podczas próby utworzenia egzemplarza klasy `ElectricCar`. Konieczne jest również uaktualnienie modułu `Car`, aby zawierał jedynie klasę `Car`.

*Plik `car.py`:*

---

```
"""Klasa, która będzie używana do zaprezentowania samochodu."""
```

```
class Car:  
    --cięcie--
```

---

Teraz możemy już importować oba wymienione moduły oddzielnie i tworzyć żądane egzemplarze samochodów.

*Plik `my_cars.py`:*

---

```
from car import Car  
from electric_car import ElectricCar  
  
my_mustang = Car('ford', 'mustang', 2024)  
print(my_mustang.get_descriptive_name())  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())
```

---

Importujemy klasę `Car` z jej modułu, a klasę `ElectricCar` z zupełnie innego modułu. Następnie tworzymy samochody: pierwszy o napędzie tradycyjnym, natomiast drugi o napędzie elektrycznym. Oba egzemplarze samochodów zostają prawidłowo utworzone:

---

```
2024 Ford Mustang  
2024 Nissan Leaf
```

---

## Używanie aliasów

Jak widziałeś w rozdziale 8., aliasy mogą być bardzo użyteczne podczas stosowania modułów do organizacji kodu źródłowego projektu. Aliasy można wykorzystać w trakcie importowania klas.

W ramach przykładu przeanalizujemy program przeznaczony do utworzenia wielu obiektów reprezentujących samochody o napędzie elektrycznym. Niestanne wpisywanie i odczytywanie nazwy klasy ElectricCar bardzo szybko stanie się uciążliwe. Dlatego w poleceniu importującym tę klasę można zdefiniować dla niego alias, jak pokazałem w kolejnym poleceniu.

```
from electric_car import ElectricCar as EC
```

Teraz zdefiniowanego aliasu można używać podczas tworzenia egzemplarza reprezentującego samochód o napędzie elektrycznym.

```
my_leaf = EC('nissan', 'leaf', 2024)
```

Alias można utworzyć także dla modułu. W kolejnym przykładzie pokazałem, jak zimportować cały moduł `electric_car` jako alias.

```
import electric_car as ec
```

W tym momencie ten alias modułu może być używany razem z nazwą klasy.

```
my_leaf = ec.ElectricCar('nissan', 'leaf', 2024)
```

## Określenie swojego sposobu pracy

Jak mogłeś zobaczyć, Python daje spory wybór w zakresie stosowanej struktury kodu w ogromnych projektach. Ważne jest poznanie wszystkich dostępnych możliwości, aby ustalić, która z nich najlepiej sprawdzi się podczas organizacji projektów. Ponadto ta wiedza pomoże Ci zrozumieć struktury stosowane w projektach przygotowanych przez innych programistów.

Kiedy rozpoczynasz pracę, postaraj się zachować jak najprostszą strukturę kodu. Pracę zaczynaj od umieszczania wszystkiego w jednym pliku, a następnie przenoś klasy do oddzielnych modułów, gdy wszystko działa zgodnie z oczekiwaniemi. Jeżeli lubisz sposób współpracy modułów i plików, spróbuj przechowywać klasy w modułach już od momentu rozpoczęcia pracy nad projektem. Znайдź podejście, które najbardziej Ci odpowiada, i stosuj je.

### ZRÓB TO SAM

**9.10. Zimportowana klasa Restaurant.** Przenieś do modułu ostatnią wersję klasy Restaurant i utwórz oddzielnny plik importujący tę klasę. Utwórz egzemplarz klasy Restaurant, a następnie wywołaj metodę tej klasy, aby sprawdzić, czy polecenie import działa prawidłowo.

**9.11. Zimportowana klasa Admin.** Pracę rozpocznij od ćwiczenia 9.8. W jednym module umieść klasy User, Privileges i Admin. Przygotuj oddzielny plik, utwórz w nim egzemplarz klasy Admin i wywołaj metodę show\_privileges(), aby sprawdzić, czy wszystko działa prawidłowo.

**9.12. Wiele modułów.** Klasę User umieść w jednym module, natomiast Privileges i Admin w oddzielnym. Następnie w innym pliku utwórz egzemplarz klasy Admin i wywołaj metodę show\_privileges(), aby sprawdzić, czy wszystko działa prawidłowo.

## Biblioteka standardowa Pythona

*Biblioteka standardowa Pythona* to zestaw modułów dostarczanych wraz z każdą instalacją języka Python. Skoro poznaleś podstawy działania klas, możesz zacząć używać modułów opracowanych przez innych programistów. Zyskujesz możliwość wykorzystania dowolnej funkcji lub klasy z biblioteki standardowej dzięki umieszczeniu prostego polecenia import na początku pliku. Spojrzymy teraz na moduł random, który może być użyteczny podczas modelowania wielu rzeczywistych obiektów.

Jedną z interesujących funkcji tego modułu jest randint(). Pobiera ona argumenty w postaci dwóch liczb całkowitych i zwraca losowo wybraną liczbę mieszczącą się w zakresie tworzonym przez argumenty funkcji.

Spójrz na sposób wygenerowania losowej liczby z przedziału od 1 do 6.

---

```
>>> from random import randint
>>> randint(1, 6)
3
```

---

Kolejną użyteczną funkcją jest choice(). Pobiera ona listę lub krotkę, a następnie zwraca losowo wybrany z niej element.

---

```
>>> from random import choice
>>> players = ['karol', 'martyna', 'michał', 'florian', 'ela']
>>> first_up = choice(players)
>>> first_up
'florian'
```

---

Moduł random nie powinien być stosowany w aplikacjach, w których zapewnienie bezpieczeństwa jest kluczową kwestią. Natomiast jest on wystarczająco dobry do użycia w wielu interesujących projektach.

**UWAGA** *Moduły można pobierać także z zewnętrznych źródeł. Wiele z nich poznasz w części drugiej książki, w której zewnętrzne moduły będą nam potrzebne do ukończenia każdego projektu.*

## ZRÓB TO SAM

**9.13. Kości do gry.** Przygotuj klasę Die z jednym atrybutem o nazwie sides, którego wartością domyślną będzie 6. Utwórz metodę o nazwie roll\_die(), wyświetlającą losowo wygenerowaną liczbę z zakresu od 1 do wartości określonej przez liczbę ścianek na kości do gry. Utwórz kość zawierającą sześć ścianek i zasymuluj rzucenie nią 10 razy.

Później utwórz kości zawierające 10 i 20 ścianek. Każdą nową kością rzuć 10 razy.

**9.14. Loteria.** Utwórz listę lub krotkę zawierającą serię dziesięciu liczb i pięciu liter. Losowo wybierz cztery liczby lub litery z listy, a następnie wyświetl komunikat informujący, że kupon zawierający liczby lub litery dopasowane do wylosowanych wygrywa nagrodę.

**9.15. Analiza loterii.** Wykorzystaj pętlę do ustalenia, jak trudno jest wygrać w loterii, którą modelowałeś w poprzednim ćwiczeniu. Utwórz listę lub krotkę o nazwie my\_ticket. Następnie zdefiniuj pętlę losującą liczby dopóty, dopóki nie zostaną dopasowane do Twojego kuponu. Wyświetl komunikat informujący, ile iteracji pętli trzeba było wykonać, zanim padły liczby znajdujące się na Twoim kuponie.

**9.16. Moduł Pythona tygodnia.** Jednym z doskonałych zasobów pomagających w poznaniu biblioteki standardowej Pythona jest witryna *Python Module of the Week*. Przejdz na stronę <https://pymotw.com/3/> i przejrzyj spis treści. Wybierz moduł, który wygląda interesująco, i przeczytaj informacje na jego temat. Przejrzyj również dokumentację dotyczącą modułu random.

## Nadawanie stylu klasom

Warto wyjaśnić kilka kwestii związanych z nadawaniem stylu klasom. To będzie ważne zwłaszcza, gdy tworzone przez Ciebie programy staną się znacznie bardziej skomplikowane.

Nazwa klasy powinna stosować styl *CamelCase*: każde słowo nazwy pisane wielką literą, brak znaków podkreślenia. Z kolei nazwy egzemplarzy klas i modułów powinny być zapisywane małymi literami, a wykorzystywanie w tych nazwach znaków podkreślenia między słowami jest dopuszczalne.

Każda klasa powinna mieć tuż po definicji komentarz typu *docstring*. W tym komentarzu należy umieścić krótki opis działania danej klasy, a zastosowane formatowanie powinno odpowiadać konwencjom używanym podczas tworzenia komentarzy typu *docstring* w funkcjach. Również dla każdego modułu należy przygotować komentarz *docstring*, tym razem wyjaśniający przeznaczenie klas znajdujących się w danym module.

Do organizacji kodu można wykorzystać puste wiersze, ale nie wolno ich nadużywać. W kodzie klasy umieszczaj pusty wiersz między metodami, natomiast klasy w module rozdzielaj dwoma pustymi wierszami.

Jeżeli musisz zaimportować moduł z biblioteki standardowej oraz moduł utworzony przez siebie, najpierw umieść polecenie importujące moduł z biblioteki standardowej. Następnie wstaw pusty wiersz i dopiero wtedy importuj samodzielnie utworzone moduły. W programach zawierających wiele poleceń `import` przedstawiona konwencja ułatwia ustalenie źródła pochodzenia różnych modułów używanych w danym programie.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak definiować własne klasy. Nauczyłeś się przechowywać informacje w klasie za pomocą atrybutów oraz tworzyć metody zapewniające klasom oczekiwane zachowanie. Zobaczyłeś, jak przygotować metodę `__init__()` odpowiedzialną za utworzenie egzemplarzy na podstawie danej klasy, zawierających dokładnie takie atrybuty, jakich oczekujesz. Poznałeś sposoby modyfikowania atrybutów egzemplarza — bezpośrednio lub za pomocą metod. Dowiedziałeś się, że dziedziczenie może znacznie ułatwić proces tworzenia powiązanych ze sobą klas, a także zobaczyłeś, jak wykorzystać egzemplarze jednej klasy jako atrybuty w innej klasie, co pozwala zachować czytelną strukturę klas.

Nauczyłeś się umieszczać klasy w modułach oraz importować klasy do pliku programu, gdy zajdzie taka potrzeba. W ten sposób będziesz mógł zapewnić swoim projektom elegancką organizację. Zacząłeś także poznawać bibliotekę standartową Pythona oraz przeanalizowałeś przykład oparty na module `random`. Na końcu dowiedziałeś się o zaleceniach dotyczących nadawania klasom stylu zgodnego z konwencjami Pythona.

W rozdziale 10. zaczniesz pracę z plikami i dzięki temu będziesz mógł zapisać pracę wykonaną w programie, a także pracę możliwą do wykonania przez użytkowników. Ponadto poznasz *wyjątki*, czyli specjalne klasy Pythona zaprojektowane po to, aby pomóc programistom odpowiednio reagować na błędy zgłasiane przez programy.

# 10

## Pliki i wyjątki



KIEDY O PANUJESZ JUŻ PODSTAWOWE UMIEJĘTNOŚCI NIEZBĘDNE DO TWORZENIA ZORGANIZOWANYCH I ŁATWYCH W UŻYCIU PROGRAMÓW, POWINIEś NAUCZYĆ SIĘ ZAPEWNIAĆ OPRACOWYWANYM PROGRAMOM JESZCZE WIĘKSZĄ UŻYTECZNOŚĆ. W tym rozdziale dowiesz się, jak pracować z plikami, których wykorzystanie pozwoli tworzonym przez Ciebie programom szybciej analizować duże ilości danych.

Nauczysz się obsługiwać błędy, aby programy nie ulegały awarii w przypadku wystąpienia nieoczekiwanych sytuacji. Poznasz *wyjątki*, czyli specjalne obiekty Pythona przeznaczone do zarządzania błędami występującymi w trakcie działania programu. Ponadto dowiesz się nieco o module json, za pomocą którego można zapisywać dane użytkownika, aby nie zostały utracone, kiedy program zakończy działanie.

Możliwość pracy z plikami oraz zapisywania danych spowoduje, że tworzone przez Ciebie programy staną się łatwiejsze w użyciu. Użytkownik będzie mógł zdecydować, kiedy i jakie dane chce podać. Ponadto zyska możliwość uruchamiania programu, wykonywania pewnych zadań, kończenia działania programu, a później wznowiania pracy od miejsca, w którym poprzednio ją zakończył. Z kolei umiejętność obsługi wyjątków pomaga w sytuacji, w której na przykład żądany plik nie istnieje lub występują inne problemy prowadzące do awarii programu. Dzięki wyjątkom programy stają się bardziej odporne na napotkanie nieprawidłowości w danych, niezależnie od tego, czy są one skutkiem zwykłej pomyłki czy złośliwą próbą złamania programu. Kiedy zdobędziesz umiejętności przedstawione w tym rozdziale, będziesz mógł tworzyć lepsze, użyteczniejsze i stabilniejsze programy.

# Odczytywanie danych z pliku

Wręcz niezwykła ilość danych znajduje się w plikach tekstowych. Tego rodzaju pliki mogą zawierać dane dotyczące pogody, ruchu drogowego, ekonomii, dzieła literackie itd. Możliwość odczytu danych z pliku jest szczególnie użyteczna w aplikacjach zajmujących się analizą danych, choć będzie również mile widziana w różnych innych sytuacjach, gdy zajdzie potrzeba przeanalizowania lub zmodyfikowania informacji przechowywanych w pliku. Przykładowo możesz utworzyć program, który będzie odczytywał zawartość pliku tekstowego, a następnie zapisywał ten plik ponownie, ale z formatowaniem pozwalającym przeglądarce internetowej wyświetlić przechowywaną w nim treść.

Kiedy chcesz pracować z informacjami znajdującymi się w pliku tekstowym, pierwszym krokiem jest wczytanie pliku do pamięci. Masz tutaj dwie możliwości. Jedna to wczytanie całej zawartości pliku, natomiast druga to praca z plikiem wiersz po wierszu.

## Wczytywanie całego pliku

Aby rozpocząć pracę, potrzebujemy pliku zawierającego kilka wierszy tekstu. Utworzymy więc plik tekstowy przechowujący liczbę pi zapisaną z dokładnością do 30 miejsc po przecinku dziesiętnym, po dziesięć cyfr w każdym wierszu.

*Plik pi\_digits.txt:*

---

```
3.1415926535
8979323846
2643383279
```

---

Jeżeli chcesz wypróbować przedstawiane tutaj przykłady, wprowadź powyższe wiersze w edytorze i zapisz plik pod nazwą *pi\_digits.txt*, lub też pobierz gotowy plik z materiałów przygotowanych do tej książki, które znajdziesz pod adresem [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Plik umieść w tym samym katalogu, w którym będą znajdowały się programy utworzone w tym rozdziale.

Poniżej przedstawiłem program otwierający podany plik oraz wyświetlający jego zawartość na ekranie.

*Plik file\_reader.py:*

---

```
from pathlib import Path

path = Path('pi_digits.txt') ❶
contents = path.read_text() ❷
print(contents)
```

---

Aby pracować z zawartością pliku, Pythonowi trzeba wskazać ścieżkę dostępu do niego. *Ścieżka dostępu* (ang. *path*) określa dokładne położenie pliku lub katalogu w systemie. Python oferuje moduł o nazwie *pathlib*, ułatwiający pracę

z plikami i katalogami, niezależnie od systemu operacyjnego, z którego korzysta użytkownik. Moduł dostarczający określoną funkcjonalność, taką jak wspomniana, jest często nazywany *biblioteką* (ang. *library*), stąd nazwa modułu `pathlib`.

Rozpoczynamy od zimportowania klasy `Path` z modułu `pathlib`. Wskazujący plik obiekt `Path` ma ogromne możliwości. Na przykład przed przystąpieniem do pracy z plikiem można sprawdzić, czy on istnieje, odczytać jego zawartość bądź też zapisać w nim nowe dane. W omawianym przykładzie tworzymy reprezentujący plik `pi_digits.txt` obiekt `Path`, który następnie przypisujemy zmiennej `path` (patrz wiersz ①). Ponieważ ten plik został zapisany w tym samym katalogu, w którym znajduje się tworzony plik programu (z rozszerzeniem `.py`), aby uzyskać do niego dostęp, obiekt `Path` wymaga jedynie nazwy pliku.

**UWAGA** *Edytor tekstu VS Code szuka plików w katalogu, który był ostatnio używany. Jeżeli korzystasz z VS Code, po uruchomieniu tego programu przejdź do katalogu zawierającego pliki dla tego rozdziału. Na przykład jeśli zapisujesz pliki programów w katalogu o nazwie rozdział\_10, naciśnij klawisze Ctrl+O (Command+O w macOS) i otwórz wymieniony katalog.*

Kiedy przygotujemy obiekt `Path` reprezentujący plik `pi_digits.txt`, w wierszu ② programu użyjemy metody `read_text()`, aby wczytać całą zawartość pliku i umieścić ją w pojedynczym, długim ciągu tekstowym o nazwie `contents`. Po wyświetleniu wartości ciągu tekstowego `contents` otrzymamy z powrotem cały plik tekstowy:

---

```
3.1415926535  
8979323846  
2643383279
```

---

Jedyna różnica między powyższymi danymi wyjściowymi oraz pierwotnym plikiem polega na dodatkowym pustym wierszu na końcu wygenerowanych danych wyjściowych. Ten wiersz znajduje się tutaj, ponieważ metoda `read_text()` zwraca pusty串tekstowy po dotarciu do końca pliku. Wspomniany pusty串tekstowy jest wyświetlany w postaci pustego wiersza na końcu wygenerowanych danych wyjściowych.

Jeżeli chcesz się pozbyć tego pustego wiersza, możesz użyć funkcji `rstrip()` w poleceniu `print()`:

---

```
from pathlib import Path  
  
path = Path('pi_digits.txt')  
contents = path.read_text()  
contents = contents.rstrip()  
print(contents)
```

---

Przypomnij sobie, że oferowana przez Pythona metoda `rstrip()` powoduje usunięcie wszystkich białych znaków znajdujących się po prawej stronie danego

ciągu tekstowego. Po tej drobnej zmianie otrzymane dane wyjściowe dokładnie odpowiadają zawartości pierwotnego pliku:

---

```
3.1415926535  
8979323846  
2643383279
```

---

Podczas wczytywania zawartości pliku można usunąć znajdujący się na końcu znak nowego wiersza. W tym celu metodę `rstrip()` należy wywołać natychmiast po metodzie `read_text()`:

```
contents = path.read_text().rstrip()
```

---

To polecenie nakazuje Pythonowi wywołanie `read_text()` w aktualnie używanym pliku. Następnie dla ciągu tekstowego zwróconego przez metodę `read_text()` zostanie wywołana metoda `rstrip()`. Oczyszczony ciąg tekstowy zostanie przypisany zmiennej `contents`. Takie podejście jest nazywane *łączeniem metod* i będzie się z nim często spotykać w programowaniu.

## Względna i bezwzględna ścieżka dostępu do pliku

Kiedy obiekowiowi `Path` zostaje przekazana prosta nazwa pliku, taka jak `pi_digits.txt`, Python szuka wskazanego pliku w katalogu, w którym znajduje się aktualnie wykonywany program, czyli plik z rozszerzeniem `.py`.

Czasami, w zależności od sposobu organizacji pracy, plik przeznaczony do otworzenia nie będzie znajdował się w tym samym katalogu, w którym jest plik programu. Przykładowo pliki programów będziesz przechowywać w katalogu `projekty_python`, natomiast pliki tekstowe używane przez programy umieścisz w znajdującym się w nim podkatalogu `pliki_tekstowe`, aby tym samym oddzielić je od programów. Pomimo tego, że podkatalog `pliki_tekstowe` znajduje się w katalogu `projekty_python`, przekazanie do obiektu `Path` nazwy pliku z podkatalogu `pliki_tekstowe` jest niewystarczające. Python ograniczy się po prostu do przeszukania jedynie katalogu `projekty_python` i na tym zakończy operację wyszukiwania — nie będzie sprawdzał żadnych podkatalogów. Aby umożliwić Pythonowi otworzenie pliku z katalogu innego niż ten, w którym znajduje się aktualnie uruchomiony program, konieczne będzie podanie poprawnej ścieżki dostępu.

Istnieją dwa podstawowe sposoby na określanie ścieżki dostępu w programowaniu. *Względna ścieżka dostępu* nakazuje Pythonowi sprawdzenie lokalizacji względem katalogu, w którym znajduje się aktualnie wykonywany program. Ponieważ podkatalog `pliki_tekstowe` znajduje się w katalogu `projekty_python`, konieczne jest przygotowanie ścieżki dostępu rozpoczynającej się od katalogu `pliki_tekstowe` i kończącej nazwą pliku. Zobacz, jak utworzyć taką ścieżkę dostępu:

```
path = Path('pliki_tekstowe/nazwa_pliku.txt')
```

---

Istnieje jeszcze możliwość podania Pythonowi dokładnej lokalizacji pliku w systemie, niezależnej od lokalizacji pliku wykonywanego programu. Mówimy wówczas o *bezwzględnej ścieżce dostępu*. Wspomnianej bezwzględnej ścieżki dostępu można użyć, gdy względna ścieżka dostępu nie działa. Jeśli na przykład umiesciszt podkatalog *pliki\_tekstowe* w zupełnie innym katalogu niż *projekty\_python*, to przekazanie do obiektu Path ścieżki dostępu *pliki\_tekstowe/nazwa\_pliku.txt* nie będzie działać, ponieważ Python sprawdzi jedynie lokalizację w katalogu *projekty\_python*. Konieczne będzie wtedy podanie pełnej ścieżki dostępu, aby Python dokładnie wiedział, gdzie szukać podanego pliku.

Bezwzględna ścieżka dostępu zwykle jest dłuższa niż względna ścieżka dostępu, ponieważ rozpoczyna się od katalogu głównego w systemie plików:

```
path = Path('/home/nazwa_użytkownika/inne_pliki/pliki_tekstowe/nazwa_pliku.txt')
```

Używając bezwzględnych ścieżek dostępu, możesz odczytywać pliki umieszczone w dowolnym miejscu systemu. Teraz jednak najłatwiejsze będzie przechowywanie odczytywanych plików w tym samym katalogu, w którym znajduje się uruchamiany program, lub też w jego podkatalogu, na przykład w podkatalogu *pliki\_tekstowe* umieszczonym w katalogu *projekty\_python*.

**UWAGA** *Podczas wyświetlania ścieżek dostępu w systemie Windows zamiast ukośnika / jest używany ukośnik \. Mimo tego w kodzie źródłowym Pythona należy stosować ukośniki /, nawet w Windowsie. Podczas pracy z dowolnym systemem operacyjnym biblioteka pathlib będzie automatycznie używala poprawnego sposobu przedstawienia ścieżki dostępu.*

## Odczytywanie wiersz po wierszu

Podczas odczytywania pliku często zachodzi potrzeba jego przeanalizowania wiersz po wierszu. Być może szukasz określonych informacji w pliku lub chcesz zmodyfikować jego tekst w konkretny sposób. Przykładowo odczytując plik z danymi dotyczącymi prognozy pogody, szukasz wiersza zawierającego słowo *słonecznie* w opisie prognozy na dany dzień. Z kolei w pliku z wiadomościami sportowymi możesz szukać wiersza zawierającego znacznik *<headline>*, aby go wyświetlić po zastosowaniu określonego rodzaju formatowania.

Dlatego też pętlę *for* można wykorzystać do iteracji przez obiekt pliku i do przeprowadzenia jego analizy wiersz po wierszu.

*Plik file\_reader.py:*

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text() ❶
```

```
lines = contents.splitlines() ❷
for line in lines:
    print(line)
```

---

Rozpoczynamy od odczytania pełnej zawartości pliku, podobnie jak wcześniej ❶. Jeżeli planujesz pracę z poszczególnymi wierszami pliku, podczas odczytywania jego zawartości nie musisz usuwać żadnych białych znaków. Metoda `splitlines()` zwraca listę wszystkich wierszy pliku. W omawianym przykładzie ta lista została przypisana zmiennej `lines` ❷. Następnie przeprowadzamy iterację przez poszczególne wiersze i je wyświetlamy:

---

```
3.1415926535
8979323846
2643383279
```

---

Ponieważ nie został zmodyfikowany żaden z tych wierszy, wygenerowane dane wejściowe dokładnie odpowiadają zawartości pierwotnego pliku tekstowego.

## Praca z zawartością pliku

Kiedy plik zostanie wczytany do pamięci, na jego zawartości można wykonywać dowolne operacje. Dlatego też pokrótce przeanalizujemy cyfry liczby pi. Zaczniemy od próby zbudowania pojedynczego ciągu tekstuowego zawierającego wszystkie cyfry wczytane z pliku, bez białych znaków.

*Plik `pi_string.py`:*

---

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines: ❸
    pi_string += line

print(pi_string)
print(len(pi_string))
```

---

Rozpoczynamy od otworzenia pliku i umieszczenia jego wierszy na liście, podobnie jak miało to miejsce w poprzednim przykładzie. Następnym krokiem jest utworzenie zmiennej `pi_string`, przeznaczonej do przechowywania cyfr liczby pi. Następnie definiujemy pętlę, w której każdy wiersz cyfr zostanie dodany do zmiennej `pi_string` ❸. Następnie wyświetlamy przygotowany ciąg tekstowy oraz liczbę wskazującą jego wielkość:

Zmienna `pi_string` zawiera białe znaki, które były po lewej stronie każdego wiersza. Możemy się ich pozbyć, używając funkcji `lstrip()` w poszczególnych wierszach:

---

```
--cięcie--
for line in lines:
    pi_string += line.lstrip()

print(pi_string)
print(len(pi_string))
```

---

Tym razem otrzymujemy ciąg tekstowy zawierający liczbę pi podaną z dokładnością do 30 miejsc po przecinku. Wielkość ciągu tekstopowego wynosi 32 znaki, ponieważ obejmuje także początkową cyfrę 3 i kropkę po niej:

---

```
3.141592653589793238462643383279
32
```

---

**UWAGA** *Kiedy Python odczytuje dane z pliku tekstopowego, cały znajdujący się w nim tekst interpretuje jako ciąg tekstopowy. Jeżeli wczytasz takie dane i chcesz pracować z nimi jak z wartościami liczbowymi, zwykle musisz skonwertować dane do postaci liczby całkowitej za pomocą funkcji `int()` lub do postaci liczby zmiennoprzecinowej za pomocą funkcji `float()`.*

## Ogromne pliki, czyli na przykład milion cyfr

Jak dotąd koncentrowaliśmy się na analizie pliku tekstopowego zawierającego jedynie trzy wiersze, ale kod zaprezentowany w tych przykładach doskonale poradzi sobie również ze znacznie większymi plikami. Jeżeli rozpocznesz pracę z plikiem tekstopowym zawierającym liczbę pi obliczoną z dokładnością do miliona miejsc po przecinku, a nie jedynie do 30 (jak wcześniej), możesz utworzyć pojedynczy ciąg tekstopowy zawierający wszystkie te cyfry. Nie ma konieczności modyfikowania programu, może poza przekazaniem mu innego pliku. W przedstawionym poniżej programie wyświetlamy jedynie pierwsze 50 miejsc po przecinku, aby nie oglądać miliona cyfr przewijanych w terminalu.

*Plik `pi_string.py`:*

---

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()
```

```
lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f'{pi_string[:52]}...')
print(len(pi_string))
```

---

Wygenerowane dane wyjściowe pokazują, że faktycznie mamy do czynienia z ciągiem tekstowym zawierającym liczbę pi podaną z dokładnością do miliona cyfr po przecinku:

---

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

---

Python nie ma narzuconego ograniczenia wskazującego maksymalną ilość danych, z jaką może pracować. Jedynym praktycznym ograniczeniem będzie ilość pamięci w systemie, którą można przeznaczyć na pracę z danymi.

**UWAGA** *Aby wypróbować ten przykład (oraz wiele innych przedstawionych w tym rozdziale), musisz pobrać materiały przygotowane do tej książki i dostępne na stronie [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/).*

## Czy data Twoich urodzin znajduje się w liczbie pi?

Zawsze byłem ciekaw, czy data moich urodzin występuje wśród cyfr tworzących liczbę pi. Przygotujemy teraz program pozwalający na sprawdzenie, czy podana data urodzenia występuje gdziekolwiek w pierwszym milionie cyfr liczby pi. W tym celu sprawdzaną datę urodzenia należy wyrazić w postaci ciągu tekstu, a później wystarczy już tylko sprawdzić, czy ten ciąg tekstowy występuje gdziekolwiek w zawartości zmiennej `pi_string`:

---

```
--cięcie--
for line in lines:
    pi_string += line.lstrip()

birthday = input("Podaj datę urodzenia (w formacie ddmmrr): ")
if birthday in pi_string:
    print("Twoja data urodzenia znajduje się wśród miliona pierwszych cyfr liczby pi!")
else:
    print("Twoja data urodzenia nie znajduje się wśród miliona pierwszych cyfr
          liczby pi.")
```

---

Na początku prosimy użytkownika o podanie daty urodzenia. Następnie sprawdzamy, czy podany ciąg tekstowy znajduje się w `pi_string`. Wypróbujmy opracowany program:

---

```
Podaj datę urodzenia (w formacie ddmmrr): 120372
Twoja data urodzenia znajduje się wśród miliona pierwszych cyfr liczby pi!
```

---

Jak widać moja data urodzenia pojawia się wśród pierwszego miliona cyfr liczby pi! Kiedy odczytamy zawartość pliku, możemy ją przeanalizować w praktycznie dowolny sposób — ograniczeni jesteśmy jedynie naszą wyobraźnią.

## ZRÓB TO SAM

**10.1. Poznajemy Pythona.** W edytorze tekstu utwórz pusty plik i wpisz w nim kilka zdań podsumowujących to, czego się dotąd nauczyłeś o języku Python. Każdy wiersz rozpocznij od wyrażenia „W Pythonie można...”. Plik zapisz pod nazwą `learning_python.txt` w tym samym katalogu, w którym umieszczasz ćwiczenia z tego rozdziału. Przygotuj program odczytujący powyższy plik i dwukrotnie wyświetlający jego zawartość. Treść pliku powinna być wyświetlona raz przez odczytanie całego pliku oraz raz przez umieszczenie wierszy na liście, a następnie przetworzenie listy za pomocą pętli.

**10.2. Poznajemy C.** Metodę `replace()` można wykorzystać do zastąpienia dowolnego słowa w ciągu tekstowym zupełnie innym słowem. Oto krótki przykład pokazujący, jak w zdaniu słowo `pies` zastąpić słowem `kot`:

---

```
>>> message = "Moje ulubione zwierzę to pies."
>>> message.replace('pies', 'kot')
'Moje ulubione zwierzę to kot.'
```

---

Odczytaj każdy wiersz tekstu w utworzonym wcześniej pliku `learning_python.txt`, a następnie wszystkie wystąpienia słowa `Python` zastąp nazwą innego języka programowania, na przykład C. Każdy zmodyfikowany wiersz wyświetl na ekranie.

**10.3. Prostszy kod.** Przedstawiony w tym podrozdziale program `file_reader.py` korzysta ze zmiennej tymczasowej `lines`, aby zaprezentować sposób działania metody `splitlines()`. Można pominać tę zmienną i przeprowadzić iterację bezpośrednio na liście zwróconej przez `splitlines()`.

---

```
for line in contents.splitlines():
```

---

Usuń zmienną tymczasową ze wszystkich programów zamieszczonych w tych podrozdziale, aby dzięki temu stały się zwięzlsze.

# Zapisywanie danych w pliku

Jednym z najprostszych sposobów zachowania danych jest ich zapis do pliku. Kiedy zapisujesz tekst w pliku, dane wyjściowe nadal będą dostępne po zamknięciu okna terminala, w którym zostały wygenerowane przez program. Dlatego też te dane wyjściowe można przeanalizować już po zakończeniu działania programu, a także udostępnić pliki danych wyjściowych innym użytkownikom. Ponadto zyskujemy możliwość tworzenia programów, które wczytują tekst z powrotem do pamięci, aby móc z nim później pracować.

## Zapisywanie pojedynczego wiersza

Po zdefiniowaniu ścieżki dostępu dane można zapisać do pliku za pomocą metody `write_text()`. Aby zobaczyć, jak to działa w praktyce, przygotujemy teraz prosty komunikat i zapiszemy go w pliku, zamiast wyświetlić na ekranie.

*Plik write\_message.py:*

---

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("Uwielbiam programować.")
```

---

Metoda `write_text()` pobiera pojedynczy argument w postaci ciągu tekstu-wego przeznaczonego do zapisania we wskazanym pliku. Omawiany program nie powoduje wygenerowania jakichkolwiek danych wyjściowych w powloce. Jeśli jednak otworzysz plik *programming.txt*, zobaczyś w nim poniższy wiersz kodu.

*Plik programming.txt:*

---

```
Uwielbiam programować.
```

---

Ten plik zachowuje się dokładnie tak samo jak każdy inny plik w komputerze. Dlatego też możesz go otwierać, dodawać do niego nowy tekst, kopiować z niego tekst, wklejać w nim tekst itd.

**UWAGA** W pliku tekstowym Python może zapisywać jedynie ciągi tekstowe. Jeżeli chcesz w pliku tekstowym umieścić wartości liczbowe, najpierw musisz je skonwertować do postaci ciągu tekstowego za pomocą funkcji `str()`.

## Zapisywanie wielu wierszy

W tle metoda `write_text()` wykonuje jeszcze kilka innych zadań. Jeżeli plik wskazywany przez `path` nie istnieje, to zostanie utworzony. Ponadto metoda gwarantuje poprawne zamknięcie pliku po przeprowadzeniu operacji zapisu. Jeżeli plik nie zostanie poprawnie zamknięty, może to prowadzić do uszkodzenia danych.

W celu zapisania do pliku więcej niż jednego wiersza trzeba przygotować ciąg tekstowy wraz z pełną zawartością pliku, a następnie wywołać metodę `write_text()` razem z argumentem w postaci tego ciągu tekstopowego. W omawianym przykładzie do pliku `programming.txt` zapisujemy kilka wierszy.

---

```
from pathlib import Path

contents = "Uwielbiam programować.\n"
contents += "Uwielbiam tworzyć gry.\n"
contents += "Uwielbiam pracować z danymi.\n"

path = Path('programming.txt')
path.write_text(contents)
```

---

W kodzie została zdefiniowana zmienna o nazwie `contents`, która służy do przechowywania pełnej zawartości pliku. W kolejnym wierszu operator `+=` pozwala dołączyć nową wartość do istniejącej zawartości zmiennej. Tę operację można powtórzyć dowolną liczbę razy i tym samym utworzyć ciąg tekstowy o dowolnej wielkości. W omawianym przykładzie na końcu każdego wiersza został dodany znak nowego wiersza, aby każda kolejna wartość była wyświetlona w oddzielnym wierszu.

Po uruchomieniu programu i otwarciu pliku `programming.txt` zobaczyś następujący blok tekstu zawierający połączone zdania:

---

```
Uwielbiam programować.
Uwielbiam tworzyć gry.
Uwielbiam pracować z danymi.
```

---

Do formatowania danych wyjściowych możesz użyć spacji, tabulatorów i pustych wierszy, podobnie jak ma to miejsce podczas formatowania danych wyjściowych w powłoce. Nie ma żadnego ograniczenia dotyczącego długości ciągu tekstopowego, a także liczby dokumentów wygenerowanych przez komputer.

**UWAGA** *Zachowaj ostrożność podczas wywoływania metody `write_text()` w obiekcie ścieżki dostępu. Jeżeli wskazany plik już istnieje, wówczas to wywołanie bieżącą zawartość pliku nadpisze nową. W dalszej części rozdziału dowiesz się, jak za pomocą biblioteki `pathlib` sprawdzać, czy plik już istnieje.*

## ZRÓB TO SAM

**10.4. Gość.** Utwórz program, który poprosi użytkownika o podanie imienia. Gdy użytkownik je wprowadzi, zapisz to imię w pliku o nazwie `guest.txt`.

**10.5. Księga gości.** Utwórz pętlę `while`, w której każdy użytkownik będzie musiał podać swoje imię. Gdy użytkownik je wprowadzi, na ekranie wyświetli komunikat powitania, a do pliku o nazwie `guest_book.txt` dodaj wiersz rejestrujący odwiedzenie Twojej strony przez tego użytkownika. Upewnij się, że każdy wpis jest umieszczany w nowym wierszu w pliku.

# Wyjątki

Do zarządzania błędami, które mogą się pojawić w trakcie wykonywania programu, Python używa specjalnych obiektów nazywanych *wyjątkami*. Gdy Python nie wie, co należy dalej zrobić z błędem, który się pojawił, wówczas tworzy obiekt wyjątku. Jeżeli utworzysz kod odpowiedzialny za obsługę wyjątku, program będzie kontynuował działanie. Natomiast jeśli nie zapewnisz obsługi zgłoszonego wyjątku, działanie programu zostanie przerwane i zostanie wyświetlony tak zwany *stos wywołań*, który zawiera informacje o zgłoszonym wyjątku.

Wyjątki są obsługiwane za pomocą bloków try-except. Blok try-except nakazuje Pythonowi wykonanie pewnego zadania, a jednocześnie mówi, co należy zrobić w przypadku zgłoszenia wyjątku. Kiedy używasz bloków try-except, program będzie kontynuował działanie nawet po wystąpieniu błędu. Zamiast wspomnianego wcześniej stosu wywołań, który może być niezrozumiały dla użytkownika, program wyświetli przyjazny użytkownikowi komunikat przygotowany przez programistę.

## Obsługiwanie wyjątku ZeroDivisionError

Zajmiemy się teraz prostym problemem, który spowoduje zgłoszenie wyjątku przez Pythona. Prawdopodobnie wiesz, że nie można dzielić liczb przez zero, ale mimo wszystko zleciemy Pythonowi wykonanie tego rodzaju zadania.

Plik division\_calculator.py:

---

```
print(5/0)
```

---

Oczywiście Python nie może wykonać tego zadania, więc wyświetla stos wywołań:

---

```
Traceback (most recent call last):
  File "division.py", line 1, in <module>
    print(5/0)
           ^
ZeroDivisionError: division by zero ①
```

---

Błąd został zgłoszony w wierszu ① stosu wywołań, a ZeroDivisionError to obiekt wyjątku. Python utworzył tego rodzaju obiekt w odpowiedzi na sytuację, w której nie umiał wykonać zleconego zadania. Kiedy wystąpi taka sytuacja, Python zatrzymuje działanie programu i wyświetla informacje o rodzaju zgłoszonego wyjątku. Następnie te informacje można wykorzystać do zmodyfikowania programu. Zyskujemy możliwość wskazania Pythonowi, co należy zrobić po wystąpieniu takiego wyjątku. Dzięki temu, jeśli ten sam wyjątek zostanie zgłoszony w przyszłości, program będzie przygotowany na jego obsługę.

## Używanie bloku try-except

Kiedy obawiasz się wystąpienia błędu, możesz przygotować blok `try-except` przeznaczony do obsługi wyjątku, który mógłby zostać zgłoszony. Nakazujesz Pythonowi wykonanie pewnego fragmentu kodu oraz informujesz go, co należy zrobić, jeśli wynikiem będzie określony rodzaj wyjątku.

Poniżej przedstawiłem przykładowy blok `try-except` przeznaczony do obsługi wyjątku `ZeroDivisionError`:

---

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("Nie można dzielić przez zero!")
```

---

Wiersz kodu `print(5/0)`, czyli wiersz powodujący wystąpienie błędu, umieściliśmy w bloku `try`. Jeżeli kod znajdujący się w bloku `try` działa, Python pomija blok `except`. Natomiast w przypadku wystąpienia błędu podczas wykonywania kodu znajdującego się w bloku `try`, Python szuka bloku `except` zawierającego obsługę błędu dopasowanego do tego zgłoszonego w bloku `try`.

W omawianym przykładzie w bloku `try` zgłaszanym jest wyjątek `ZeroDivisionError`, więc Python szuka dopasowanego dla niego bloku `except`, aby wiedzieć, w jaki sposób ma zareagować. Następnie Python wykonuje kod znajdujący się w tym bloku, a użytkownik zamiast tajemniczego stosu wywołań otrzymuje zrozumiałą dla siebie komunikat błędu:

---

Nie można dzielić przez zero!

---

Jeżeli po konstrukcji `try-except` znajduje się dodatkowy kod, program będzie kontynuował działanie, ponieważ wskazaliśmy Pythonowi sposób obsługi błędu. Przechodzimy teraz do przykładu, w którym przechwycenie błędu pozwala programowi kontynuować działanie.

## Używanie wyjątków w celu uniknięcia awarii programu

Prawidłowa obsługa błędów jest szczególnie ważna, gdy program ma do wykonania więcej pracy po wystąpieniu błędu. Tego rodzaju sytuacja zdarza się dość często w programach, w których użytkownik musi wprowadzić pewne dane wejściowe. Gdy program odpowiednio zareaguje na nieprawidłowe dane wejściowe, wówczas może poprosić użytkownika o podanie poprawnych danych, zamiast po prostu doprowadzić do awarii.

Utworzmy teraz prosty kalkulator przeprowadzający jedynie operację dzielenia.

### Plik division\_calculator.py:

---

```
print("Podaj dwie liczby, które zostaną podzielone.")
print("Wpisz 'q', aby zakończyć działanie programu.")

while True:
    first_number = input("\nPierwsza liczba: ") ❶
    if first_number == 'q':
        break
    second_number = input("Druga liczba: ") ❷
    if second_number == 'q':
        break
    answer = int(first_number) / int(second_number) ❸
    print(answer)
```

---

W wierszu ❶ program prosi użytkownika o podanie pierwszej liczby, która będzie przechowywana w zmiennej `first_number`. Jeśli użytkownik nie wpisze `q`, aby tym samym zakończyć działanie programu, zostanie poproszony również o podanie drugiej liczby, która będzie przechowywana w zmiennej `second_number` (patrz wiersz ❷). Podzielenie podanych liczb daje nam oczekiwany wynik, który zostaje zapisany w zmiennej `answer` (patrz wiersz ❸). Ten program nie zawiera żadnego kodu przeznaczonego do obsługi błędów, więc próba dzielenia przez zero doprowadzi do awarii:

---

```
Podaj dwie liczby, które zostaną podzielone.
Wpisz 'q', aby zakończyć działanie programu.
```

```
Pierwsza liczba: 5
Druga liczba: 0
Traceback (most recent call last):
  File "division.py", line 11, in <module>
    answer = int(first_number) / int(second_number)
                ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

---

Aвария програму то кіпське розв'язання, зрештю подобнє як відображення на екрані комп'ютера. Позбавлені ведомостей технічної користувача і так ніч не зрозумієть з цих інформації, жодні потенційні атакуючі можуть на цьому засновані. На приклад познайомлюємося з ім'ям файлу програми, а також дізнаємося, яка частина цієї програми не працює правильно. Досвідчений атакуючий може часом використати цю інформацію, щоб зробити правильну атаку на код програми.

## Blok `else`

Програма може стати більшою на помилки, якщо він виконується, коли може привести до появи помилки, обернімши конструкцію `try-except`. В описаному прикладі помилка виникає в відповідності операції ділення,

więc właśnie tam umieścimy blok try-except. W kodzie znalazł się również blok else. Każdy fragment kodu, którego wykonanie zależy od zakończonego powodzeniem wykonania kodu w bloku try, powinien znaleźć się w bloku else:

```
--cięcie--  
while True:  
    --cięcie--  
    if second_number == 'q':  
        break  
    try: ❶  
        answer = int(first_number) / int(second_number)  
    except ZeroDivisionError: ❷  
        print("Nie można dzielić przez zero!")  
    else: ❸  
        print(answer)
```

W bloku try (patrz wiersz ❶) nakazujemy Pythonowi przeprowadzenie operacji dzielenia. Tutaj znajduje się tylko kod, którego wykonanie może doprowadzić do wygenerowania błędu. Pozostałe wiersze kodu, których wykonanie zależy od zakończonego powodzeniem wykonania bloku try, zostały umieszczone w bloku else. W omawianym przykładzie, jeśli operacja dzielenia zakończy się sukcesem, blok else wykorzystamy do wyświetlenia wyniku (patrz wiersz ❸).

Blok except wskazuje Pythonowi, jak należy zareagować w przypadku zgłoszenia wyjątku ZeroDivisionError (patrz wiersz ❷). Jeżeli wykonanie bloku try zakończyło się niepowodzeniem z powodu próby dzielenia przez zero, wyświetlamy użytkownikowi zrozumiałą dla niego komunikat o tym, jak ma uniknąć tego rodzaju błędu. Program kontynuuje działanie, a użytkownik nigdy nie zobaczy stosu wywołań:

---

Podaj dwie liczby, które zostaną podzielone.  
Wpisz 'q', aby zakończyć działanie programu.

Pierwsza liczba: 5  
Druga liczba: 0  
Nie można dzielić przez zero!

Pierwsza liczba: 5  
Druga liczba: 2  
2.5

Pierwsza liczba: q

---

W bloku try powinien się znaleźć jedynie ten fragment kodu, który może spowodować zgłoszenie wyjątku. Czasami mamy jeszcze dodatkowy kod przeznaczony do wykonania tylko wtedy, gdy wykonanie bloku try zakończy się sukcesem. Ten dodatkowy kod należy umieścić w bloku else. Z kolei blok except wskazuje

Pythonowi, co trzeba zrobić w przypadku zgłoszenia wyjątku podczas wykonywania kodu zdefiniowanego w bloku `try`.

Dzięki określeniu potencjalnych źródeł błędów można tworzyć znacznie bardziej niezawodne programy, które będą kontynuowały działanie nawet po napotkaniu nieprawidłowych danych lub w przypadku braku wymaganych zasobów. Ponadto kod stanie się odporniejszy na przypadkowe błędy popełniane przez użytkowników lub celowe próby przeprowadzenia złośliwych ataków na program.

## Obsługa wyjątku `FileNotFoundException`

Jeden z często występujących problemów podczas pracy z plikami wiąże się z obsługą brakujących plików. Wymagany plik może znajdować się w innej lokalizacji, nazwa pliku mogła zostać błędnie podana lub plik w ogóle nie istnieje. Wszystkie tego rodzaju sytuacje można w prosty sposób obsłużyć za pomocą konstrukcji `try-except`.

Spróbujmy więc odczytać zawartość nieistniejącego pliku. Przedstawiony poniżej program próbuje wczytać treść *Alicji w Krainie Czarów*, ale plik `alice.txt` nie znajduje się w tym samym katalogu, w którym mamy program `alice.py`.

Plik `alice.py`:

---

```
from pathlib import Path

path = Path('alice.txt')
contents = path.read_text(encoding='utf-8')
```

---

Zwróć uwagę na użycie wywołania `read_text()` w nieco innej postaci niż we wcześniejszych przykładach. Argument `encoding` jest niezbędny, gdy domyślne kodowanie znaków w systemie nie odpowiada kodowaniu znaków w otwieranym pliku. Z taką sytuacją mamy najczęściej do czynienia, gdy odczytywany plik nie został utworzony w danym systemie.

Python nie może odczytać danych z nieistniejącego pliku, więc zostaje zgłoszony wyjątek:

---

```
Traceback (most recent call last):
  File "alice.py", line 4, in <module> ①
    contents = path.read_text(encoding='utf-8') ②
              ~~~~~
  File ".../pathlib.py", line 1056, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
              ~~~~~
  File ".../pathlib.py", line 1042, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
              ~~~~~
FileNotFoundException: [Errno 2] No such file or directory: 'alice.txt' ③
```

---

To jest nieco dłuższy stos wywołań niż prezentowane wcześniej, więc możesz zobaczyć, jak przedstawiają się bardziej złożone stopy wywołań. Jego analizę często najlepiej będzie rozpocząć od samego końca. W ostatnim wierszu znajduje się informacja o zgłoszonym wyjątku `FileNotFoundException` ❸. To jest bardzo ważne, ponieważ wskazuje rodzaj wyjątku, który powinien zostać użyty w bloku kodu `except`.

W pobliżu początku stosu wywołań ❶ zobaczyś, że błąd wystąpił w wierszu 4. pliku `alice.py`. W następnym wierszu został podany numer wiersza kodu, który spowodował wygenerowanie błędu ❷. Pozostała część stosu wywołań przedstawia wybrany kod bibliotek użyty podczas otwierania pliku i odczytywania jego zawartości. Z reguły nie trzeba dokładnie analizować tych wszystkich wierszy stosu wywołań.

Aby obsłużyć zgłoszony błąd, należy w bloku `try` umieścić wiersz wskazany przez stos wywołań jako problematyczny. W omawianym przykładzie będzie to wiersz zawierający wywołanie `read_text()`.

---

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError: ❶
    print(f"Przepraszamy, ale plik {path} nie istnieje.")
```

---

W omawianym przykładzie kod znajdujący się w bloku `try` powoduje wygenerowanie błędu `FileNotFoundException`, więc Python szuka bloku `except` dopasowanego do tego błędu ❶. Następnie wykonuje kod zdefiniowany w bloku `except`, w wyniku czego użytkownik zamiast tajemniczego stosu wywołań otrzymuje zrozumiałą dla siebie komunikat błędu:

---

```
Przepraszamy, ale plik alice.txt nie istnieje.
```

---

Jeżeli plik nie istnieje, to program nie ma nic więcej do zrobienia i dlatego kod odpowiedzialny za obsługę błędu ogranicza swoje działanie jedynie do wyświetlenia komunikatu o braku pliku. Przechodzimy więc do rozbudowanej wersji tego przykładu. Zobaczysz, jak obsługa błędów może być pomocna podczas pracy z więcej niż tylko jednym plikiem.

## Analiza tekstu

Z pomocą Pythona możesz analizować pliki tekstowe zawierające całe książki. Wiele klasycznych dzieł literatury jest dostępnych w postaci zwykłych plików tekstowych, ponieważ są uznawane za własność publiczną. Teksty wykorzystane w tym rozdziale pochodzą z projektu Gutenberg (<http://www.gutenberg.org/>). Projekt Gutenberg zawiera kolekcję dzieł literatury klasycznej dostępnej jako

własność publiczna — jest to doskonaly zasób, gdy szukasz dzieł, które mógłbyś wykorzystać w projektach programistycznych.

W tym rozdziale użyjemy tekstu książki *Alicja w Krainie Czarów* i obliczymy liczbę słów znajdujących się w tym utworze. Podczas pracy wykorzystamy metodę o nazwie `split()`, która potrafi podzielić ciąg tekstowy w miejscu wystąpienia dowolnego białego znaku.

---

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print(f"Przepraszamy, ale plik {path} nie istnieje.")
else:
    # Obliczenie przybliżonej liczby słów w pliku.
    words = contents.split() ❶
    num_words = len(words) ❷
    print(f"Plik {path} zawiera {num_words} słów.")
```

---

Plik *alice.txt* umieszczamy teraz w odpowiednim katalogu, aby tym razem kod znajdujący się w bloku `try` został wykonany prawidłowo. Pobieramy ciąg tekstowy `contents`, który obecnie zawiera cały tekst utworu *Alicja w Krainie Czarów* w postaci pojedynczego, długiego ciągu tekstopowego. Następnie za pomocą metody `split()` generujemy listę wszystkich słów w tym tekście ❶. Kolejnym krokiem jest użycie funkcji `len()` względem przygotowanej listy, aby ustalić jej wielkość. Otrzymujemy dość dokładną liczbę słów znajdujących się w pierwotnym ciągu tekstopowym (patrz wiersz ❷). Na końcu wyświetlamy komunikat informujący o liczbie słów znalezionych w pliku. Ten fragmentu kodu został umieszczony w bloku `else`, ponieważ będzie działał tylko wtedy, gdy wykonanie kodu bloku `try` zakończy się sukcesem.

Wygenerowane dane wyjściowe pokazują, ile słów znajduje się w pliku *alice.txt*:

---

```
Plik alice.txt zawiera 29461 słów.
```

---

Podana liczba jest nieco zawyżona, ponieważ obejmuje również dodatkowe informacje umieszczone w pliku tekstopowym przez wydawcę. Jednak mimo wszystko otrzymujemy przybliżoną liczbę słów w utworze *Alicja w Krainie Czarów*.

## Praca z wieloma plikami

Dodamy teraz inne książki do przeanalizowania. Jednak zanim to zrobimy, większość logiki programu przeniesiemy do funkcji o nazwie `count_words()`. W ten sposób znacznie ułatwigimy sobie zadanie przeanalizowania wielu książek.

## Plik word\_count.py:

---

```
from pathlib import Path

def count_words(path):
    """Obliczenie przybliżonej liczby słów w danym pliku.""" ❶
    try:
        contents = path.read_text(encoding='utf-8')
    except FileNotFoundError:
        print(f'Przepraszamy, ale plik {path} nie istnieje.')
    else:
        # Obliczenie przybliżonej liczby słów w pliku.
        words = contents.split()
        num_words = len(words)
        print(f"Plik {path} zawiera {num_words} słów.")

path = Path('alice.txt')
count_words(path)
```

---

Większość przedstawionego powyżej kodu pozostała niezmieniona. Po prostu zastosowaliśmy wcięcia i kod odpowiedzialny za obliczenie liczby słów przeniesliśmy do funkcji o nazwie `count_words()`. Dobrym nawykiem jest uaktualnianie komentarzy, kiedy modyfikujemy program, więc zmieniliśmy nieco komentarz `docstring` (patrz wiersz ❶).

Teraz możemy utworzyć prostą pętlę ustalającą liczbę słów w dowolnym pliku, którego zawartość chcemy przeanalizować. W tym celu nazwy plików przeznaczonych do analizy przechowujemy na liście, a następnie wywołujemy funkcję `count_words()` dla każdego ze wskazanych plików. W omawianym przykładzie obliczamy liczbę słów znajdujących się w następujących książkach: *Alicja w Krainie Czarów*, *Siddhartha*, *Moby Dick* i *Male kobietki*. Wszystkie wymienione powieści są uznawane za własność publiczną. Celowo umieściłem plik *siddhartha.txt* poza katalogiem zawierającym program `word_count.py`, aby pokazać, jak w programie można obsługiwać sytuację, gdy wymagany plik nie istnieje:

---

```
from pathlib import Path

def count_words(path):
    --cięcie--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    path = Path(filename) ❶
    count_words(path)
```

---

Nazwy plików są przechowywane w postaci zwykłych ciągów tekstowych. Następnie przed wywołaniem `count_words()` każdy z nich jest konwertowany na obiekt `Path` ❶. Brak pliku *siddhartha.txt* nie ma żadnego wpływu na wykonanie pozostałej części programu:

---

Plik alice.txt zawiera 29461 słów.  
Przepraszamy, ale plik siddhartha.txt nie istnieje.  
Plik moby\_dick.txt zawiera 215136 słów.  
Plik little\_women.txt zawiera 189079 słów.

---

Użycie konstrukcji `try-except` w przedstawionym powyżej programie przynosi dwie ważne korzyści. Po pierwsze, użytkownik nie zobaczy niezrozumiałego dla niego stosu wywołań. Po drugie, jeśli którykolwiek z podanych plików nie istnieje, program będzie kontynuował działanie i przeanalizuje dostępne pliki. Gdy nie przechwycimy błędu `FileNotFoundException` wygenerowanego na skutek braku pliku `siddhartha.txt`, wówczas użytkownik zobaczy niezrozumiałą dla niego stos wywołań, a sam program zakończy działanie po rozpoczęciu analizy wymienionego pliku. Nigdy nie zostanie podjęta próba przeanalizowania następnych podanych plików.

## Ciche niepowodzenie

W poprzednim przykładzie użytkownik otrzymywał komunikat, gdy jeden z plików był niedostępny. Jednak nie ma konieczności zgłoszenia każdego przechwyconego wyjątku. Czasami oczekiwane jest, aby informacja o niepowodzeniu nie była wyświetlana w przypadku zgłoszenia wyjątku, a program kontynuował działanie, jakby nic się nie stało. Jeżeli chcesz zezwolić na ciche niepowodzenie, blok `try` tworzysz w standardowy sposób, natomiast w bloku `except` wyraźnie wskazujesz Pythonowi, że nie powinien podejmować żadnych działań. Python oferuje polecenie `pass` informujące o braku konieczności podjęcia jakichkolwiek działań w danym bloku:

---

```
def count_words(path):
    """Obliczenie przybliżonej liczby słów w danym pliku."""
    try:
        --cięcie--
    except FileNotFoundError:
        pass
    else:
        --cięcie--
```

---

Jedyna różnica między powyższym i poprzednim programem polega na poleceniu `pass` w bloku `except`. Teraz po zgłoszeniu wyjątku `FileNotFoundException` nastąpi wykonanie bloku `except`, ale nic się nie stanie. Nie zostanie wyświetlony stos wywołań, nie zostaną również wyświetcone żadne komunikaty o zgłoszonym wyjątku. Użytkownicy zobaczą informacje dotyczące liczby słów w istniejących plikach, nie będzie żadnej wzmianki o jakimkolwiek niezalezionym pliku:

---

Plik alice.txt zawiera 29461 słów.  
Plik moby\_dick.txt zawiera 215136 słów.  
Plik little\_women.txt zawiera 189079 słów.

---

Polecenie `pass` działa w charakterze miejsca zarezerwowanego. Przypomina, że nie zdecydowałeś się na podjęcie żadnych działań w tym konkretnym miejscu wykonywania programu i być może zmienisz to w przyszłości. W omawianym programie możemy na przykład za jakiś czas postanowić umieścić nazwy brakujących plików w innym pliku o nazwie `missing_files.txt`. Wprawdzie użytkownicy nie będą mieli dostępu do tego pliku, ale Ty tak, i w ten sposób zyskasz możliwość zajęcia się wszelkimi brakującymi plikami.

## Które błędy należy zgłaszać?

Skąd wiadomo, kiedy należy zgłosić błąd użytkownikowi oraz czy należy zezwolić na ciche niepowodzenia? Jeżeli użytkownik dokładnie wie, które pliki powinny być przeanalizowane, wówczas może być wdzięczny za komunikat wyjaśniający, dlaczego pewne pliki nie zostały przeanalizowane. Gdy użytkownik oczekuje otrzymania pewnych wyników, choć nie wie, jakie książki będą przedmiotem analizy, wtedy niekoniecznie trzeba go informować o nieistniejących plikach. Przedstawianie użytkownikowi nieoczekiwanych przez niego informacji może spowodować zmniejszenie użyteczności danego programu. Oferowana przez Pythona struktura przeznaczona do obsługi błędów zapewnia programistom dokładną kontrolę nad ilością informacji udostępnianych użytkownikom, gdy coś pójde nie tak. Tylko od programisty zależy, ile tych informacji zostanie udostępnionych.

Doskonale napisany i prawidłowo przetestowany kod nie jest zbyt podatny na błędy wewnętrzne, takie jak błędy składni lub logiczne. Jednak w każdym przypadku, gdy działanie programu zależy od czynników zewnętrznych (na przykład danych wejściowych pobieranych od użytkownika, istnienia pliku lub dostępności połączenia sieciowego), ryzyko wystąpienia wyjątku rośnie. Nawet niewielkie doświadczenie pomoże w ustaleniu, gdzie w tworzonych programach umieszczać bloki obsługi wyjątków, a także jak wiele informacji o wygenerowanych błędach należy przedstawić użytkownikom.

### ZRÓB TO SAM

**10.6. Dodawanie.** Jeden z najczęściej pojawiających się problemów podczas pobierania danych liczbowych polega na tym, że użytkownicy podają tekst zamiast liczb. Kiedy tego rodzaju dane wejściowe spróbujesz skonwertować na typ `int`, otrzymasz błąd `ValueError`. Utwórz program, który prosi użytkownika o podanie dwóch liczb. Dodaj je i wyświetl wynik. Przechwyć błąd `ValueError`, jeżeli którakolwiek wartość przekazana w danych wejściowych nie jest liczbą, i wyświetl odpowiedni komunikat błędu. Przetestuj program, najpierw podając dwie liczby, a później pewien tekst zamiast liczby.

**10.7. Kalkulator dodawania.** Kod przygotowany w ćwiczeniu 10.6 opakuj pętlą `while`, aby użytkownik mógł kontynuować wprowadzanie liczb nawet po popełnieniu błędu i podaniu tekstu zamiast liczby.

**10.8. Koty i psy.** Utwórz dwa pliki o nazwach `cats.txt` i `dogs.txt`. W pierwszym pliku umieść przynajmniej trzy imiona kotów, natomiast w drugim przynajmniej trzy imiona psów. Przygotuj program próbujący odczytać zawartość tych plików i wyświetlić ją na ekranie. Zastosuj w kodzie konstrukcję `try-except`, aby przechwycić wszelkie błędy typu `FileNotFoundException` i wyświetlić użytkownikowi odpowiedni komunikat błędu, gdy żądany plik nie istnieje. Jeden z utworzonych wcześniej plików przenieś do innego katalogu w systemie i upewnij się, że opracowany blok `except` w programie działa prawidłowo.

**10.9. Ciche koty i psy.** Blok `except` w programie utworzonym w poprzednim ćwiczeniu zmodyfikuj w taki sposób, aby brak pliku powodował jedynie ciche niepowodzenie.

**10.10. Najczęściej występujące słowa.** Odwiedź witrynę projektu Gutenberg (<http://www.gutenberg.org/>) i wybierz kilka innych książek, które chciałbyś przeanalizować. Pobierz pliki tekstowe tych dzieł lub niezmodyfikowany zwykły tekst skopiuj z przeglądarki internetowej do pliku tekstowego w komputerze.

Za pomocą metody `count()` możesz sprawdzić, ile razy dane słowo lub wyrażenie występuje w ciągu tekstu. Na przykład w przedstawionym poniżej fragmencie kodu sprawdzamy, ile razy słowo za występuje w ciągu tekstu:

```
>>> line = "Za górami, za lasami, za dolinami pobili się dwaj górale  
ciupagami."  
>>> line.count('za')  
2  
>>> line.lower().count('za')  
3
```

Zwróć uwagę na to, że konwersja ciągu tekstu na zapisany małymi literami (do tego służy funkcja `lower()`, którą poznaleś już wcześniej) powoduje przechwycenie wszystkich wystąpień sprawdzanego słowa niezależnie od sposobu jego formatowania.

Utwórz program odczytujący pobrane wcześniej z projektu Gutenberg pliki tekstowe, a następnie określający, ile razy słowo the występuje w każdym z nich. Otrzymany wynik będzie jedynie przybliżony, ponieważ uwzględnia również słowa zawierające 'za', np. 'zachód' lub 'zabawa'. Spróbuj zliczyć wystąpienia 'za' (wraz ze spaciem w ciągu tekstu) i zobacz, o ile mniejsza będzie ta liczba.

## Przechowywanie danych

Wiele programów będzie prosiło użytkowników o podanie pewnego rodzaju informacji. Możesz pozwolić użytkownikowi przechowywać preferencje w grze, lub też dostarczać dane niezbędne do przygotowania wizualizacji. Niezależnie od przeznaczenia programu informacje dostarczane przez użytkowników są przechowywane w strukturach danych takich jak listy i słowniki. Kiedy użytkownik

zakończy działanie programu, niemal zawsze należy zachować wprowadzone przez niego informacje. Prostym rozwiązaniem, które można w tej sytuacji zastosować, jest przechowywanie danych za pomocą modułu o nazwie `json`.

Moduł `json` pozwala zapisywać w plikach proste struktury danych Pythona oraz wczytywać dane z tych plików podczas kolejnej sesji pracy z programem. Ponadto wymieniony moduł można wykorzystać do współdzielenia danych między różnymi programami Pythona. Co ważniejsze, format danych JSON nie jest charakterystyczny jedynie dla Pythona, więc zapisane w nim dane można wymieniać także z osobami tworzącymi kod w innych językach programowania. JSON to użytkuczny, przenośny i łatwy do poznania format przechowywania danych.

**UWAGA** Format JSON (JavaScript Object Notation) został pierwotnie opracowany dla języka JavaScript. Jednak od tamtej chwili stał się powszechnie stosowanym formatem w wielu językach programowania, w tym także w Pythonie.

## Używanie `json.dumps()` i `json.loads()`

Przygotujemy teraz krótki program, który będzie przechowywał zestaw liczb, oraz inny program, który wczyta te liczby do pamięci. W pierwszym programie wykorzystamy funkcję `json.dumps()` w celu zapisania zestawu liczb do pliku. Natomiast w drugim programie wykorzystamy funkcję `json.loads()` w celu wczytania tych liczb.

Funkcja `json.dumps()` pobiera jeden argument: dane przeznaczone do skonwertowania na format JSON. Ta funkcja zwraca ciąg tekstowy, który następnie można zapisać do pliku danych.

Plik `number_writer.py`:

---

```
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]

path = Path('numbers.json') ❶
contents = json.dumps(numbers) ❷
path.write_text(contents)
```

---

Na początku programu importujemy moduł `json` oraz tworzymy listę liczb, z którymi będziemy pracować. W wierszu ❶ wybieramy nazwę pliku przeznaczonego do przechowywania listy liczb. Podaliśmy tutaj rozszerzenie `.json`, aby wskazać, że dane przechowywane w tym pliku są zapisane w formacie JSON. Następnie używamy funkcji `json.dumps()` (patrz wiersz ❷) do wygenerowania ciągu tekstu zawierającego reprezentację JSON danych roboczych. Mając ten串tekstowy, można go zapisać do pliku, używając tej samej metody `write_text()`, która była wykorzystywana już wcześniej.

Omawiany program nie generuje danych wyjściowych, ale możesz otworzyć plik *numbers.json* i spojrzeć na jego zawartość. Dane są przechowywane w formacie przypominającym dane Pythona.

---

[2, 3, 5, 7, 11, 13]

---

Przystępujemy teraz do utworzenia programu wykorzystującego funkcję `json.loads()` w celu wczytania zapisanej listy z powrotem do pamięci.

*Plik number\_reader.py:*

---

```
from pathlib import Path
import json

path = Path('numbers.json') ❶
contents = path.read_text() ❷
numbers = json.loads(contents) ❸

print(numbers)
```

---

W wierszu ❶ upewniamy się, że odczyt odbywa się dokładnie z tego samego pliku, w którym wcześniej zapisaliśmy dane. Ponieważ plik danych to tak naprawdę plik zwykłego tekstu, jego zawartość można odczytać za pomocą metody `read_text()`, jak pokazałem w wierszu ❷. W wierszu ❸ zawartość pliku jest przekazywana funkcji `json.loads()`. Ta funkcja pobiera ciąg tekstowy sformatowany jako JSON i zwraca obiekt Pythona (w omawianym przykładzie jest to lista), który tutaj umieszczamy w zmiennej `numbers`. Na koniec wyświetlamy przywróconą listę liczb i sprawdzamy, czy to taka sama lista, jak utworzona w programie *number\_writer.py*:

---

[2, 3, 5, 7, 11, 13]

---

W taki prosty sposób można współdzielić dane między dwoma programami.

## Zapisywanie i odczytywanie danych wygenerowanych przez użytkownika

Zapisywanie danych za pomocą modułu `json` jest użyteczne podczas pracy z danymi wygenerowanymi przez użytkownika, ponieważ informacje te, jeśli nie zostaną gdzieś zapisane, zostaną utracone po zakończeniu działania programu. Spójrz na przykład programu, w którym prosimy użytkownika o podanie imienia w trakcie pierwszego uruchomienia programu. To imię zostaje zapisane i jest używane w trakcie kolejnych uruchomień programu.

Rozpoczynamy od zachowania w pliku imienia użytkownika.

### Plik remember\_me.py:

---

```
from pathlib import Path
import json

username = input("Jak masz na imię? ") ❶

path = Path('username.json') ❷
contents = json.dumps(username)
path.write_text(contents)

print(f"Twoje imię zostało zapisane i będzie używane później, {username}!") ❸
```

---

W wierszu ❶ prosimy użytkownika o podanie imienia, które zostanie zapisane w celu późniejszego użycia. Następnie otrzymane dane zapisujemy w pliku *username.json* (patrz wiersz ❷). Następnym krokiem jest wyświetlenie użytkownikowi komunikatu o zapisaniu jego imienia (patrz wiersz ❸):

---

```
Jak masz na imię? Eryk
Twoje imię zostało zapisane i będzie używane później, Eryk!
```

---

Teraz przechodzimy do utworzenia nowego programu, w którym użytkownik zostanie przywitany z użyciem imienia wcześniejszo zapisanego w pliku.

### Plik greet\_user.py:

---

```
from pathlib import Path
import json

path = Path('username.json') ❶
contents = path.read_text()
username = json.loads(contents) ❷

print(f"Witamy ponownie, {username}!)
```

---

W wierszu ❶ za pomocą funkcji `json.loads()` odczytujemy imię zapisane w pliku *username.json* i umieszczaemy je w zmiennej `username` ❷. Po przywróceniu imienia można je wykorzystać do przywitania użytkownika:

---

```
Witamy ponownie, Eryk!
```

---

Dwa przedstawione powyżej programy warto połączyć w jednym pliku. Kiedy uruchomimy program *remember\_me.py*, pobierze on imię z pamięci, o ile zostało tam umieszczone. W przeciwnym razie program poprosi użytkownika o podanie imienia i zapisze je w pliku *username.json* w celu użycia w trakcie następnego uruchomienia programu. W prawdziwej sytuacji tutaj użyć bloku `try-except`, aby

zapewnić odpowiednią reakcję programu, jeżeli plik `username.json` jeszcze nie istnieje, ale zamiast tego skorzystamy z użytecznej metody modułu `pathlib`.

*Plik remember\_me.py:*

---

```
from pathlib import Path
import json

path = Path('username.json')
if path.exists(): ❶
    contents = path.read_text()
    username = json.loads(contents)
    print(f"Witamy ponownie, {username}!")
else: ❷
    username = input("Jak masz na imię? ")
    contents = json.dumps(username)
    path.write_text(contents)
    print(f"Twoje imię zostało zapisane i będzie używane później, {username}!")
```

---

Mamy wiele przydatnych metod, których można używać razem z obiektami `Path`. Metoda `exists()` zwraca wartość `True`, jeśli istnieje wskazany plik lub katalog. W przeciwnym razie wartością zwrotną jest `False`. W kodzie użyliśmy wywołania `path.exists()`, aby ustalić, czy imię użytkownika zostało już zapisane w pliku ❶. Jeżeli ten plik istnieje, odczytujemy zapisane w nim imię i wyświetlamy komunikat powitania użytkownika.

Natomiast plik `username.json` jeszcze nie istnieje (patrz wiersz ❷), prosimy użytkownika o podanie imienia i zapisujemy je do pliku. Ponadto wyświetlamy komunikat informujący, że podane imię zostanie później użyte.

Niezależnie od tego, który blok kodu został wykonany, wynikiem jest zachowanie imienia do późniejszego użycia i wyświetlenie odpowiedniego powitania. W przypadku pierwszego uruchomienia programu dane wyjściowe wyglądają następująco:

---

Jak masz na imię? Eryk

Twoje imię zostało zapisane i będzie używane później, Eryk!

---

Jeśli użytkownik uruchamia program po raz kolejny, otrzymujemy poniższe dane wyjściowe:

---

Witamy ponownie, Eryk!

---

Te dane wyjściowe zostaną wygenerowane, gdy program został już uruchomiony przynajmniej jeden raz. Wprawdzie dane użyte w tym przykładzie to pojedynczy ciąg tekstowy, ale program będzie działał równie dobrze z dowolnymi danymi, które zostały skonwertowane na postać ciągu tekstowego w formacie JSON.

## Refaktoryzacja

Bardzo często docieramy do punktu, w którym kod działa, choć jednocześnie mamy świadomość, że można go jeszcze bardziej usprawnić przez podział na serię funkcji wykonujących konkretne zadania. Taki proces jest określany mianem *refaktoryzacji*. Dzięki refaktoryzacji kod staje się bardziej przejrzysty, prostszy do zrozumienia oraz łatwiejszy do dalszej rozbudowy.

Refaktoryzację programu *remember\_me.py* możemy przeprowadzić przez przeniesienie większości jego logiki do co najmniej jednej funkcji. Zadaniem omawianego programu jest przywitanie użytkownika, więc cały istniejący kod przenosimy do funkcji o nazwie *greet\_user()*.

*Plik remember\_me.py:*

---

```
from pathlib import Path
import json

def greet_user():
    """Przywanie użytkownika z użyciem jego imienia.❶"""
    path = Path('username.json')
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        print(f'Witamy ponownie, {username}!')
    else:
        username = input("Jak masz na imię? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f'Twoje imię zostało zapisane i będzie używane później,
{username}!')

greet_user()
```

---

Ponieważ teraz będziemy używać funkcji, konieczne jest uaktualnienie komentarzy *docstring*, aby odzwierciedlały aktualny sposób działania programu (patrz wiersz ❶). Wprawdzie kod w pliku stał się bardziej przejrzysty, ale działanie funkcji *greet\_user()* nie polega jedynie na przywitaniu użytkownika. Funkcja pobiera również imię zapisane w pliku, o ile taki istnieje, lub prosi użytkownika o podanie imienia, jeśli wskazany plik jeszcze nie istnieje.

Przeprowadzimy więc refaktoryzację funkcji *greet\_user()*, aby nie wykonywała zbyt wielu różnych zadań. Pracę rozpoczętymy od przeniesienia do oddzielnej funkcji kodu odpowiedzialnego za pobranie imienia przechowywanego w pliku:

---

```
from pathlib import Path
import json

def get_stored_username(path):
```

---

```

"""Pobranie imienia z pliku, o ile taki istnieje."""
❶ if path.exists():
    contents = path.read_text()
    username = json.loads(contents)
    return username
else:
    return None ❷

def greet_user():
    """Przywitanie użytkownika z użyciem jego imienia."""
    path = Path('username.json')
    username = get_stored_username(path)
    if username: ❸
        print(f"Witamy ponownie, {username}!")
    else:
        username = input("Jak masz na imię? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"Twoje imię zostało zapisane i będzie używane później, {username}!")

greet_user()

```

---

Nowa funkcja o nazwie `get_stored_username()` jasno wskazuje swoje przeznaczenie przedstawione w komentarzu *docstring* (patrz wiersz ❶). Zadaniem tej funkcji jest pobranie imienia zapisanego w pliku i jego zwrot, o ile plik istnieje. Jeżeli plik o nazwie `username.json` nie istnieje, wartością zwrotną funkcji będzie `None` (patrz wiersz ❷). Jest to dobra praktyka: funkcja powinna zwrócić oczekiwany wartość lub `None`. Dzięki temu będzie możliwa przeprowadzić prosty test wartości zwrotnej funkcji. W wierszu ❸ wyświetlamy użytkownikowi komunikat powitania, o ile pobranie imienia z pliku zakończyło się powodzeniem. Natomiast w przypadku niepowodzenia, prosimy użytkownika o podanie imienia.

Do refaktoryzacji pozostał nam jeszcze jeden blok kodu poza funkcją `greet_user()`. Jeżeli imię nie zostało pobrane z pliku, to operacja, w której prosimy użytkownika o podanie imienia, a następnie zapisujemy je w pliku, powinna być przeprowadzona przez funkcję dedykowaną specjalnie do tego celu:

---

```

from pathlib import Path
import json

def get_stored_username(path):
    """Pobranie imienia z pliku, o ile taki istnieje."""
    --cięcie--

def get_new_username():
    """
    Poproszenie użytkownika, aby podał swoje imię,
    a następnie zapisanie tego imienia w pliku.
    """
    username = input("Jak masz na imię? ")

```

```

contents = json.dumps(username)
path.write_text(contents)
return username

def greet_user():
    """Przywitanie użytkownika z użyciem jego imienia."""
    path = Path('username.json')
    username = get_stored_username(path) ❶
    if username:
        print(f"Witamy ponownie, {username}!")
    else:
        username = get_new_username(path) ❷
        print(f"Twoje imię zostało zapisane i będzie używane później, {username}!")

greet_user()

```

---

Każda funkcja w ostatecznej wersji programu *remember\_me.py* ma jeden wyraźnie zdefiniowany cel. Wywołujemy funkcję `greet_user()` wyświetlającą odpowiedni komunikat, czyli powitanie użytkownika, który korzystał już z programu, lub też powitanie zupełnie nowego użytkownika. Odbywa się to przez wywołanie funkcji `get_stored_username()` ❶ odpowiedzialnej za pobranie imienia z pliku, o ile taki istnieje. W przypadku braku pliku z imieniem funkcja `greet_user()` wywołuje funkcję `get_new_username()` ❷, której zadanie polega na pobraniu imienia wprowadzonego przez użytkownika i na zapisaniu jego w pliku. Podział logiki na mniejsze fragmenty ma kluczowe znaczenie podczas tworzenia przejrzystego kodu, który będzie łatwy w obsłudze oraz dalszej rozbudowie.

## ZRÓB TO SAM

**10.11. Ulubiona liczba.** Utwórz program, który prosi użytkownika o podanie ulubionej liczby. Za pomocą funkcji `json.dumps()` zapisz tę liczbę w pliku. Następnie utwórz oddzielny program odczytujący ulubioną liczbę użytkownika i wyświetlający komunikat w stylu: „Znam Twoją ulubioną liczbę, to \_\_\_\_”.

**10.12. Zapamiętana ulubiona liczba.** Oba programy utworzone w poprzednim ćwiczeniu połącz w jednym pliku. Jeżeli ulubiona liczba została zapisana w pliku, wyświetl ją użytkownikowi. W przeciwnym razie poproś użytkownika o podanie ulubionej liczby i zapisz ją w pliku. Uruchom ten program dwukrotnie i upewnij się, że działa prawidłowo.

**10.13. Słownik użytkownika.** Program *remember\_me.py* przechowuje tylko jeden fragment informacji, nazwę użytkownika. Rozbuduj go w taki sposób, aby prosił o podanie jeszcze dwóch innych informacji. Następnie wszystkie dane dostarczone przez użytkownika umieść w słowniku. Za pomocą wywołania `json.dumps()` zapisz słownik do pliku, a później odczytaj go z pliku, używając wywołania `json.loads()`. Program powinien wyświetlić podsumowanie dokładnie informujące, jakie przechowuje dane dotyczące użytkownika.

**10.14. Weryfikacja użytkownika.** W ostatniej wersji programu *remember\_me.py* przyjęto założenie, że użytkownik już wcześniej podał swoje imię, lub też program został uruchomiony po raz pierwszy. Powinniśmy zmodyfikować ten program na wypadek, gdyby bieżący użytkownik nie był tą osobą, która ostatnio korzystała z tego programu.

W funkcji `greet_user()`, zanim za pomocą odpowiedniego komunikatu powitasz istniejącego już użytkownika, zapytaj go, czy podane imię jest poprawne. Jeżeli nie jest, należy wywołać funkcję `get_new_username()` w celu użycia prawidłowego imienia.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak pracować z plikami. Nauczyłeś się odczytywać jednorazowo całą zawartość pliku, a także odczytywać treść pliku wiersz po wierszu. Poznałeś sposoby zapisywania dowolnej ilości danych w pliku. Ponadto poznaleś wyjątki i zobaczyłeś, jak obsługiwać je w tworzonych programach. Na końcu rozdziału dowiedziałeś się, jak przechowywać struktury danych Pythona, aby móc zapisywać informacje wprowadzane przez użytkowników. Dzięki temu użytkownicy nie będą musieli podawać tych danych przy każdym uruchomieniu programu.

W rozdziale 11. poznasz efektywne sposoby testowania kodu. Przeprowadzanie testów pomaga się upewnić, że opracowany kod jest prawidłowy. Ponadto ułatwia wykrywanie błędów wprowadzanych podczas dalszego rozbudowywania programów.

# 11

## Testowanie kodu



KIEDY UTWORZYSZ FUNKCJĘ LUB KLASĘ, DLA ZDEFINIOWANEGO W NIEJ KODU MOŻESZ PRZYGOTOWAĆ TAKŻE TEST. DZIĘKI TESTOM MOŻNA SIĘ UPEWNIĆ, ŻE KOD DZIAŁA ZGODNIE Z OCZEKIWANIAMI W PRZYPADKU wszystkich rodzajów otrzymywanych danych wejściowych. Jeżeli przygotujesz testy, będziesz mieć pewność, że kod będzie działał prawidłowo nawet wtedy, gdy coraz więcej osób zacznie z niego korzystać. Ponadto nowo dodany kod będzie można od razu przetestować i upewnić się, że nie spowoduje uszkodzenia istniejącej funkcjonalności. Każdy programista popełnia błędy i dlatego każdy programista powinien często testować kod oraz wychwytywać problemy, zanim zostaną one odkryte przez użytkowników.

W tym rozdziale dowiesz się, jak przetestować tworzony kod Pythona za pomocą narzędzi oferowanych przez bibliotekę pytest. Ta biblioteka to zbiór narzędzi pomagających szybko i łatwo utworzyć pierwsze testy, a jednocześnie oferuje duże możliwości w zakresie obsługi testów, gdy staną się one bardziej skomplikowane w projektach. Python domyślnie nie zawiera biblioteki pytest, więc wyjaśnię sposób instalacji bibliotek zewnętrznych. Dzięki takiej umiejętności zyskujesz dostęp do ogromnej ilości doskonale opracowanego kodu, który można wykorzystać we własnych projektach. Te biblioteki ogromnie zwiększają zakres projektów, nad którymi możesz pracować.

Nauczysz się tworzyć zestawy testów oraz sprawdzać, czy dane wejściowe skutkują otrzymaniem oczekiwanych danych wyjściowych. Zobaczysz, jak wygląda zaliczenie testu, niepowodzenie testu oraz jak niezaliczony test pomaga w usprawnieniu tworzonego kodu. Poznasz sposoby testowania funkcji i klas, a także nauczysz się ustalać, jaką liczbę testów należy przygotować dla danego projektu.

# Instalowanie pytest za pomocą pip

Wprawdzie biblioteka standardowa Pythona oferuje ogromną ilość funkcjonalności, ale programiści Pythona intensywnie korzystają także z *pakietów opracowanych przez podmioty zewnętrzne*. Są to biblioteki opracowane poza podstawowym pakietem Pythona. Niektóre z najpopularniejszych bibliotek zewnętrznych ostatecznie trafiły do biblioteki standardowej i od tamtej chwili zaczęły być dodawane do większości instalacji Pythona. Tak się dzieje najczęściej w przypadku bibliotek, które zbytnio się już nie zmieniają po usunięciu ich początkowych błędów. Tego rodzaju biblioteki ewoluują w tym samym tempie co język.

Jednak wiele pakietów jest dostępnych poza biblioteką standardową, więc mogą być rozwijane niezależnie od samego języka. Takie pakiety zwykle są uaktualniane znacznie częściej, niż byłyby w przypadku ich włączenia do harmonogramu prac nad Pythonem. To dotyczy pytest i większości pozostałych bibliotek, które zostaną użyte w drugiej części książki. Nie należy ufać bezgranicznie każdemu pakietowi opracowanemu przez podmioty zewnętrzne. Oczywiście nie należy również zrażać się tym, że ważna funkcjonalność została zaimplementowana za pomocą pakietu zewnętrznego.

## Uaktualnianie pip

Python zawiera narzędzie o nazwie pip, używane do instalowania pakietów zewnętrznych. Ponieważ pip pomaga w instalacji pakietów opracowanych przez podmioty zewnętrzne, jest często uaktualniany w celu usuwania potencjalnych luk w zabezpieczeniach. Dlatego trzeba zacząć od uaktualnienia tego narzędzia.

Otwórz nowe okno powłoki i wydaj następujące polecenie:

---

```
$ python -m pip install --upgrade pip
Requirement already satisfied: pip in /.../python3.11/site-packages (22.0.4) ❶
--cięcie--
Successfully installed pip-22.1.2 ❷
```

---

Pierwsza część polecenia, `python -m pip`, nakazuje Pythonowi uruchomienie modułu pip. Natomiast druga część, `install --upgrade`, nakazuje uaktualnienie już zainstalowanego pakietu. Ostatnia część polecenia, `pip`, wskazuje pakiet zewnętrzny przeznaczony do uaktualnienia. Wygenerowane dane wyjściowe pokazują, że bieżąca wersja pip, 22.0.4 ❶, została zastąpiona najnowszą dostępną wersją, 22.1.2 ❷.

Następujące polecenie można wykorzystać do uaktualnienia dowolnego pakietu zewnętrznego, który jest zainstalowany w systemie:

---

```
$ python -m pip install --upgrade nazwa_pakietu
```

---

**UWAGA** Jeżeli używasz systemu Linux, narzędzie pip może nie być dołączone do Twojej instalacji Pythona. Dlatego jeśli podczas próby uaktualnienia narzędzia pip otrzymasz komunikat błędu, zapoznaj się z informacjami zamieszczonymi w dodatku A.

## Instalowanie pytest

Skoro narzędzie pip zostało zaktualizowane, można przystąpić do instalacji biblioteki pytest.

```
$ python -m pip install --user pytest
Collecting pytest
  --cięcie--
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x
```

Tym razem polecenie `pip install` zostało użyte bez opcji `--upgrade`. Zamiast niej wykorzystaliśmy opcję `--user`, nakazującą Pythonowi zainstalowanie tego pakietu jedynie dla bieżącego użytkownika. Wygenerowane przez to polecenie dane wyjściowe wskazują na zakończoną sukcesem operację instalacji najnowszej dostępnej wersji pakietu `pytest`, razem z innymi pakietami zależnymi dla `pytest`.

Oto polecenie, które można wykorzystać do zainstalowania dowolnego pakietu zewnętrznego:

```
$ python -m pip install --user nazwa_pakietu
```

**UWAGA** Jeżeli napotkasz trudności podczas wykonywania tego polecenia, spróbuj je wydać bez opcji `--user`.

## Testowanie funkcji

Aby dowiedzieć się nieco na temat testowania, najpierw trzeba przygotować kod, który później będzie testowany. Poniżej przedstawiłem prostą funkcję pobierającą imię i nazwisko, a następnie zwracającą elegancko sformatowane pełne imię i nazwisko.

Plik `name_function.py`:

```
def get_formatted_name(first, last):
    """Generuje elegancko sformatowane pełne imię i nazwisko."""
    full_name = f'{first} {last}'
    return full_name.title()
```

Funkcja `get_formatted_name()` łączy imię i nazwisko, umieszczając między nimi spację, aby w ten sposób powstało pełne imię i nazwisko. Następnie pierwsze litery imienia i nazwiska są zamieniane na wielkie. Jeżeli chcesz sprawdzić, jak działa powyższa funkcja, po prostu utwórz wykorzystujący ją program. Poniżej przedstawiłem program `names.py`, w którym użytkownik po podaniu imienia i nazwiska otrzymuje elegancko sformatowane pełne imię i nazwisko.

*Plik names.py:*

---

```
from name_function import get_formatted_name

print("Wpisz 'q', aby zakończyć działanie programu.")
while True:
    first = input("\nPodaj imię: ")
    if first == 'q':
        break
    last = input("Podaj nazwisko: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tElegancko sformatowane pełne imię i nazwisko: {formatted_name}.")
```

---

Ten program importuje funkcję `get_formatted_name()` z pliku modułu `name_function.py`. Użytkownik może podać imię i nazwisko, a otrzyma elegancko sformatowane pełne imię i nazwisko, jak pokazałem w poniższym przykładzie:

---

Wpisz 'q', aby zakończyć działanie programu.

Podaj imię: **janis**

Podaj nazwisko: **joplin**

Elegancko sformatowane pełne imię i nazwisko: Janis Joplin.

Podaj imię: **bob**

Podaj nazwisko: **dylan**

Elegancko sformatowane pełne imię i nazwisko: Bob Dylan.

Podaj imię: **q**

---

Wyraźnie widać, że wygenerowane dane wyjściowe są prawidłowe. Przyjmujemy założenie, że chcemy zmodyfikować funkcję `get_formatted_name()` w taki sposób, aby obsługiwała także drugie imię. Oczywiście chcemy zachować pewność, że w żaden sposób nie zostanie naruszona funkcjonalność programu, gdy dana osoba będzie miała tylko jedno imię i nazwisko. Wprawdzie można przetestować kod przez uruchomienie programu `names.py` i wprowadzanie danych takich jak `Janis Joplin` po każdej modyfikacji funkcji `get_formatted_name()`, ale takie podejście bardzo szybko stanie się uciążliwe. Na szczęście pytest zapewnia efektywny sposób automatyzacji testowania danych wyjściowych funkcji. Jeżeli

zautomatyzujemy testowanie funkcji `get_formatted_name()`, zawsze będziemy mieli pewność, że działa prawidłowo dla tego rodzaju imienia i nazwiska, dla którego przygotowaliśmy testy.

## Test jednostkowy i zestaw testów

Istnieje wiele różnych podejść w zakresie testowania oprogramowania. Jednym z najprostszych jest tzw. test jednostkowy. *Test jednostkowy* sprawdza poprawność jednego konkretnego aspektu zachowania funkcji. Z kolei *zestaw testów* to kolekcja testów jednostkowych, które łącznie mają potwierdzić, że działanie danej funkcji jest zgodne z oczekiwaniemi w szerokiej gamie sytuacji, w których wystąpienia tej funkcji można się spodziewać.

Dobry zestaw testów jednostkowych obejmuje wszystkie możliwe rodzaje danych wejściowych, jakie funkcja może otrzymać, oraz zawiera testy dostosowane do każdej sytuacji. Zestaw testów o pełnym pokryciu zawiera pełny zakres testów wykorzystujących wszystkie możliwe sposoby, na jakie można używać funkcji. W przypadku dużych projektów zapewnienie pełnego pokrycia testami może być zmudne. Bardzo często wystarczy przygotować testy dla szczególnie istotnych fragmentów kodu. Dążyć do pełnego pokrycia testami można dopiero wtedy, gdy projekt zyskuje coraz większą akceptację.

## Zaliczenie testu

Dzięki pytest utworzenie pierwszego testu jednostkowego jest dość proste. Przygotujemy pojedynczy test dla naszej funkcji. Będzie on wywoływał sprawdzaną funkcję i używał asercji dotyczącej wartości zwrotnej tej funkcji. Jeżeli asercja okaże się poprawna, test zostanie zaliczony. W przypadku niepoprawnej asercji test zakończy się niepowodzeniem.

Poniżej przedstawiłem test sprawdzający poprawność działania funkcji `get_name_function()`.

*Plik test\_name\_function.py:*

---

```
from name_function import get_formatted_name

def test_first_last_name(): ❶
    """Czy dane w postaci 'Janis Joplin' są obsługiwane prawidłowo?"""
    formatted_name = get_formatted_name('janis', 'joplin') ❷
    assert formatted_name == 'Janis Joplin' ❸
```

---

Przed uruchomieniem testu warto nieco dokładniej zapoznać się z definiującą go funkcją. Nazwa pliku testu ma duże znaczenie — musi rozpoczynać się od ciągu tekstuowego `test_`. Gdy nakazujesz pytest wykonanie zdefiniowanych testów, wówczas biblioteka rozpoczęcie wyszukiwanie pliku o nazwie rozpoczęjącej się od `test_`, a następnie uruchomi wszystkie znajdujące się w nim testy.

Prace zaczynamy od zimportowania funkcji przeznaczonej do przetestowania, czyli tutaj `get_formatted_name()`. Następnie definiujemy funkcję testu,

w omawianym przykładzie jest to `test_first_last_name()` ❶. Nazwa tej funkcji jest dłuższa niż używane dotychczas i ma to swoje uzasadnienie. Przede wszystkim nazwa funkcji testu musi rozpoczynać się od słowa `test` i znaku podkreślenia. Każda funkcja o nazwie rozpoczynającej się od `test_` zostanie *odkryta* przez `pytest` i będzie wykonywana jako część procesu testowania.

Ponadto nazwy testów powinny być dłuższe i bardziej opisowe niż zwykłych funkcji. Tego rodzaju funkcji nigdy nie wywołujesz samodzielnie, to biblioteka `pytest` je wyszukuje i wykonuje za programistę. Nazwa funkcji testu powinna być na tyle opisowa, że gdy zobaczysz ją w wygenerowanym raporcie po wykonaniu testów, będziesz doskonale wiedzieć, jaki sposób działania został przetestowany.

Następnie wywołujemy funkcję przeznaczoną do przetestowania (patrz wiersz ❷). W omawianym przykładzie wywoływana jest funkcja `get_formatted_name()` wraz z argumentami '`janis`' i '`joplin`', podobnie jak w przypadku jej wykonania w programie `names.py`. Wynik jej działania zostaje umieszczony w zmiennej `formatted_name`.

W wierszu ❸ używamy asercji. Asercja sprawdza, czy otrzymany wynik odpowiada oczekiwанemu. W tym przypadku oczekujemy, że wartością zwrotną funkcji `get_formatted_name()` powinno być '`Janis Joplin`'.

## Wykonywanie testu

Jeżeli plik `test_name_function.py` uruchomisz bezpośrednio, nie otrzymasz żadnych danych wyjściowych, ponieważ funkcja testowa nigdy nie zostanie wykonana. Zamiast tego to `pytest` zajmuje się uruchomieniem pliku testowego.

W tym celu otwórz okno terminala i przejdź do katalogu zawierającego plik testu. Jeżeli używasz VS Code, w edytorze możesz otworzyć katalog zawierający plik testu, a następnie wykorzystać terminal wbudowany w VS Code. W terminalu wydaj polecenie `pytest`.

---

```
$ pytest
=====
platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x ❶
rootdir: /.../projekty_pythona/rozdzial_11 ❷
collected 1 item ❸

test_name_function.py .
=====
1 passed in 0.00s ===== [100%] ❹
```

---

Spróbujmy przeanalizować te dane wyjściowe. Przede wszystkim otrzymujemy informacje o systemie, w którym został wykonany test ❶. Używam systemu macOS, więc w tym wierszu możesz otrzymać nieco inne dane wyjściowe. Co najważniejsze, podana jest tutaj wersja Pythona, biblioteki `pytest` oraz innych pakietów używanych podczas wykonywania testu.

Następnie został podany katalog, w którym znajduje się wykonywany test ❷. W tym przypadku to *projekty\_pythona/rozdział\_11*. Widzimy, że biblioteka `pytest` znalazła tylko jeden test do wykonania ❸ i to właśnie ten plik testu został uruchomiony ❹. Kropka po nazwie pliku informuje o zaliczeniu pojedynczego testu. Zapis [100%] na końcu wiersza potwierdza wykonanie wszystkich testów. Ogromny projekt może zawierać setki lub tysiące testów, więc kropki i wskaźniki procentowe będą okazywały się niezwykle pomocne podczas monitorowania ogólnego postępu wykonywania testów.

W ostatnim wierszu mamy komunikat o wykonaniu jednego testu w czasie krótszym niż 0,01 sekundy.

Otrzymane dane wyjściowe potwierdzają, że kiedy do funkcji `get_formatted_name()` zostanie przekazane imię i nazwisko, funkcja ta zawsze będzie poprawnie działała, przynajmniej dopóki nie zostanie zmodyfikowana. Po wprowadzeniu zmian w `get_formatted_name()` należy ponownie wykonać test. Jeżeli zostanie zaliczony, nadal mamy pewność, że funkcja działa prawidłowo w przypadku przekazania jej danych takich jak Janis Joplin.

**UWAGA** Jeżeli nie masz pewności, jak z poziomu terminala przejść do odpowiedniego katalogu, zapoznaj się z informacjami zamieszczonymi w rozdziale 1. Ponadto jeśli otrzymasz komunikat informujący o nieznalezieniu polecenia `pytest`, zamiast niego wydaj polecenie `python -m pytest`.

## Niezaliczenie testu

Jak wygląda niezaliczony test? Zmodyfikujemy teraz funkcję `get_formatted_name()` tak, aby mogła obsługiwać także drugie imię. Jednak wprowadzona przez nas zmiana spowoduje uszkodzenie funkcjonalności funkcji podczas obsługi osób mających tylko jedno imię, na przykład takich jak Janis Joplin.

Poniżej przedstawiłem nową wersję funkcji `get_formatted_name()`, która tym razem wymaga argumentu w postaci drugiego imienia.

*Plik name\_function.py:*

---

```
def get_formatted_name(first, middle, last):
    """Generuje elegancko sformatowane pełne imię i nazwisko."""
    full_name = f'{first} {middle} {last}'
    return full_name.title()
```

---

Nowa wersja funkcji będzie doskonale działać dla osób posiadających drugie imię, ale kiedy ją przetestujemy, okaże się, że przy okazji wprowadzania zmian uszkodziliśmy funkcjonalność w zakresie osób mających jedynie imię i nazwisko.

Tym razem uruchomienie `test_name_function.py` powoduje wygenerowanie następujących danych wyjściowych:

---

```
$ pytest
=====
 test session starts =====
--cięcie--
```

---

```

test_name_function.py F [100%] ①
===== FAILURES ===== ②
    test_first_last_name ③
def test_first_last_name():
    """Do names like 'Janis Joplin' work?"""
>     formatted_name = get_formatted_name('janis', 'joplin') ④
E     TypeError: get_formatted_name() missing 1 required positional ⑤
        argument: 'last'
test_name_function.py:5: TypeError
===== short test summary info =====
FAILED test_name_function.py::test_first_last_name - TypeError:
    get_formatted_name() missing 1 required positional argument: 'last'
===== 1 failed in 0.04s =====

```

---

Wygenerowane dane wyjściowe zawierają wiele informacji, ponieważ w przypadku niezaliczenia testu prawdopodobnie będziesz chciał wiedzieć, dlaczego tak się stało. Pierwszy element danych wyjściowych, na który trzeba zwrócić uwagę, to pojedyncza litera F (patrz wiersz ①) informująca, że jeden test jednostkowy w zestawie testów nie został zaliczony. Następnie mamy sekcję FAILURES, koncentrującą się na niepowodzeniach (patrz wiersz ②), ponieważ niezaliczone testy są najważniejsze w danych wyjściowych wygenerowanych po wykonaniu testów. Jak można zobaczyć w wierszu ③, Python wskazuje, że błąd został spowodowany przez metodę `test_first_last_name()`. Z kolei nawias ostry (patrz wiersz ④) wskazuje wiersz kodu, który doprowadził do niezaliczenia testu. Litera E w następnym wierszu ⑤ pokazuje rzeczywisty błąd, z powodu którego doszło do niezaliczenia testu. W omawianym przykładzie jest to błąd `TypeError`, który został wygenerowany, ponieważ zabrakło wymaganego argumentu pozycyjnego `last`. Najważniejsze informacje są ponownie przedstawione w krótkim podsumowaniu na końcu, aby podczas wykonywania wielu testów programista mógł od razu poznać liczbę niezaliczonych testów, bez konieczności przewijania wielu wierszy danych wyjściowych.

## Reakcja na niezaliczony test

Co należy zrobić w przypadku niezaliczenia testu? Jeżeli przyjmiemy założenie, że test sprawdza poprawność warunków, to zaliczenie testu oznacza prawidłowe działanie funkcji, natomiast niezaliczenie wskazuje na istnienie błędu w nowo utworzonym kodzie. Kiedy więc test nie zostanie zaliczony, nie zmieniaj testu. Jeżeli to zrobisz, test może zostać zaliczony, ale kod wywołujący tę funkcję może nagle przestać działać prawidłowo, podobnie jak wcześniej test. Zamiast tego spróbuj poprawić kod, który spowodował niezaliczenie testu. Przeanalizuj zmiany wprowadzone w funkcji i postaraj się ustalić, w jaki sposób te modyfikacje mogły spowodować uszkodzenie wcześniej działających funkcjonalności.

W przypadku funkcji `get_formatted_name()` wcześniej wymagane były jedynie dwa parametry: imię i nazwisko. Po modyfikacji funkcji wymagane jest podanie imienia, drugiego imienia i nazwiska. Dodanie obowiązkowego parametru w postaci drugiego imienia spowodowało wypaczenie oczekiwanej zachowania funkcji

`get_formatted_name()`. Najlepszym rozwiązaniem będzie tutaj określenie drugiego imienia jako opcjonalnego. Wówczas test przeprowadzany dla danych wejściowych takich jak `Janis Joplin` znów będzie zaliczony, a ponadto funkcja będzie potrafiła obsługiwać drugie imię. Zmodyfikujemy teraz funkcję `get_formatted_name()` w taki sposób, aby drugie imię było opcjonalne, a następnie ponownie wykonamy zestaw testów. Jeżeli zostanie zaliczony, przejdziemy do utworzenia drugiego testu, dzięki któremu upewnimy się, że funkcja prawidłowo obsługuje również sytuację, w której podano drugie imię.

Aby drugie imię było opcjonalne, musimy przenieść parametr `middle` na koniec listy parametrów w definicji funkcji i przypisać mu wartość domyślną w postaci pustego ciągu tekstowego. Ponadto dodajemy konstrukcję `if` odpowiedzialną za prawidłowe zbudowanie pełnego imienia i nazwiska w zależności od tego, czy podano drugie imię.

Plik `name_function.py`:

---

```
def get_formatted_name(first, last, middle=''):  
    """Generuje elegancko sformatowane pełne imię i nazwisko."""  
    if middle:  
        full_name = f"{first} {middle} {last}"  
    else:  
        full_name = f"{first} {last}"  
    return full_name.title()
```

---

W nowej wersji funkcji `get_formatted_name()` drugie imię stało się opcjonalne. Kiedy zostanie przekazane do funkcji (`if middle:`), wtedy pełne imię i nazwisko będzie składało się z imienia, drugiego imienia oraz nazwiska, w przeciwnym razie — tylko imienia i nazwiska. Gdy wprowadzimy tę modyfikację, funkcja powinna działać prawidłowo z obydwoema rodzajami danych wejściowych. Aby sprawdzić, czy funkcja nadal działa zgodnie z oczekiwaniemi w przypadku podania jako danych `Janis Joplin`, ponownie uruchamiamy `test_name_function.py`:

---

```
$ pytest  
===== test session starts =====  
--cięcie--  
test_name_function.py . [100%]  
===== 1 passed in 0.00s =====
```

---

Zestaw testów zostaje zaliczony. Jest to idealna sytuacja i oznacza, że funkcja działa prawidłowo dla imion i nazwisk w postaci `Janis Joplin`, bez konieczności ręcznego testowania funkcji. Poprawienie funkcji było łatwe, ponieważ zakończony niepowodzeniem test pomógł w zidentyfikowaniu nowego kodu, którego dodanie spowodowało uszkodzenie istniejącej funkcjonalności.

## Dodanie nowego testu

Skoro już wiemy, że funkcja `get_formatted_name()` ponownie działa dla prostych imion i nazwisk, możemy utworzyć drugi test sprawdzający osoby mające drugie imię. W tym celu dodajemy kolejną funkcję testu na końcu pliku `test_name_function.py`.

Plik `test_name_function.py`:

---

```
from name_function import get_formatted_name

def test_first_last_name(self):
    --cięcie--

def test_first_last_middle_name(self):
    """Czy dane w postaci 'Wolfgang Amadeus Mozart' są obsługiwane prawidłowo?"""
    formatted_name = get_formatted_name(❶
        'wolfgang', 'mozart', 'amadeus')
    assert formatted_name == 'Wolfgang Amadeus Mozart' ❷
```

---

Nowej metodzie nadajemy nazwę `test_first_last_middle_name()`. Nazwa funkcji musi się rozpoczynać od `test_`, aby była wykonywana automatycznie po uruchomieniu `pytest`. Nazwa nowej funkcji jasno wskazuje, które zachowanie metody `get_formatted_name()` jest tutaj testowane. Dlatego też jeżeli test zakończy się niepowodzeniem, od razu będzie wiadomo, jakiego rodzaju imiona i nazwiska sprawią problem.

W celu przetestowania funkcji wywołujemy `get_formatted_name()`, podając imię, drugie imię i nazwisko (patrz wiersz ❶), a następnie używamy asercji ❷ do sprawdzenia, czy wartość zwrotna odpowiada oczekiwaniu pełnemu imienia i nazwiska w postaci imienia, drugiego imienia i nazwiska. Po wydaniu polecenia `pytest` widzimy, że oba testy zostały zaliczone:

---

```
$ pytest
=====
test session starts =====
--cięcie--
collected 2 items

test_name_function.py .. [100%] ❶
===== 2 passed in 0.01s =====
```

---

Dwie kropki (patrz wiersz ❶) wskazują na zaliczenie dwóch testów, co jest potwierdzone również w ostatnim wierszu danych wyjściowych. Doskonale! Teraz już wiemy, że funkcja nadal działa w przypadku osób o imieniu i nazwisku w stylu `Jannis Joplin`. Ponadto możemy być pewni prawidłowego działania funkcji także w przypadku osób posiadających drugie imię, na przykład `Wolfgang Amadeus Mozart`.

## ZRÓB TO SAM

**11.1. Miasto, państwo.** Przygotuj funkcję akceptującą dwa parametry: nazwy miasta i państwa. Wartością zwracaną tej funkcji powinien być pojedynczy ciąg tekstowy w postaci *Miasto, Państwo*, na przykład Santiago, Chile. Gotową funkcję umieść w module o nazwie *city\_functions.py*. Ten plik zapisz w nowym katalogu, aby biblioteka pytest nie próbowała wykonywać testów zdefiniowanych we wcześniejszej części rozdziału.

Utwórz plik o nazwie *test\_cities.py* przeznaczony do przetestowania przygotowanej wcześniej funkcji. Następnie w pliku *test\_cities.py* zdefiniuj funkcję o nazwie *test\_city\_country()* odpowiedzialną za sprawdzenie, czy wywołanie utworzonej w poprzednim ćwiczeniu funkcji, na przykład z wartościami 'santiago' i 'chile', spowoduje wygenerowanie oczekiwanej ciągi tekstowego. Uruchom plik *test\_cities.py* i upewnij się, że test *test\_city\_country()* zostaje zaliczony.

**11.2. Populacja.** Zmodyfikuj przygotowaną wcześniej funkcję, aby wymagała podania trzeciego argumentu — populacji (*population*). Teraz wartością zwracaną funkcji powinien być ciąg tekstowy w postaci *Miasto, Państwo - populacja xxx*, na przykład Santiago, Chile - populacja 5000000. Ponownie wykonaj testy. Upewnij się, że tym razem test zdefiniowany w funkcji *test\_city\_country()* będzie niezaliczony.

Zmodyfikuj funkcję, aby parametr *population* był opcjonalny. Ponownie wykonaj testy i upewnij się, że również teraz test zdefiniowany w metodzie *test\_city\_country()* zostanie zaliczony.

Utwórz drugi test o nazwie *test\_city\_country\_population()*, sprawdzający, czy można wywołać funkcję z wartościami 'santiago', 'chile' i 'population =>=5000000'. Raz jeszcze wykonaj testy i upewnij się, że nowy test został zaliczony.

# Testowanie klasy

W pierwszej części tego rozdziału utworzyłeś testy dla pojedynczej funkcji. Teraz przystąpimy do utworzenia testów dla klasy. Z klas będziesz korzystać w wielu własnych projektach, więc możliwość sprawdzenia poprawności ich działania niewątpliwie jest użyteczna. Jeżeli zostaną zaliczone testy dla klasy, nad którą pracujesz, będziesz miał pewność, że wprowadzone w niej usprawnienia nie spowodują przypadkowego uszkodzenia aktualnej funkcjonalności danej klasy.

## Różne rodzaje metod asercji

Dotychczas przedstawiłem tylko jeden rodzaj asercji — twierdzenie, że ciąg tekstowy ma określoną wartość. Podczas tworzenia testów można definiować dowolne twierdzenia, przedstawiane w postaci wyrażenia warunkowego. Jeżeli

zgodnie z oczekiwaniami zostanie wygenerowana wartość True, to założenia przyjęte co do sposobu działania programu są prawidłowe. Możesz być pewien, że nie istnieją w nim błędy. Natomiast jeśli jest zwracana wartość False zamiast oczekiwanej True, wtedy test zakończy się niepowodzeniem i wiadomo, że istnieje problem, który trzeba rozwiązać. W tabeli 11.1 wymieniłem sześć najczęściej używanych asercji, które można stosować w początkowo tworzonych testach.

Tabela 11.1. Najczęściej używane w testach polecenia asercji

Asercja	Opis
assert a == b	Przyjęcie założenia, że dwie wartości są równe.
assert a != b	Przyjęcie założenia, że dwie wartości nie są równe.
assert a	Przyjęcie założenia, że a przyjmuje wartość True.
assert not a	Przyjęcie założenia, że a przyjmuje wartość False.
assert element in lista	Przyjęcie założenia, że element jest na liście.
assert element not in lista	Przyjęcie założenia, że element nie znajduje się do listie.

To zaledwie kilka przykładów. Wszystko, co można wyrazić za pomocą konstrukcji warunkowej, można uwzględnić także w teście.

## Klasa do przetestowania

Testowanie klasy odbywa się podobnie jak testowanie funkcji — większość pracy po stronie programisty wiąże się ze sprawdzeniem zachowania metod zdefiniowanych w danej klasie. Istnieje jednak kilka różnic. Zaczynamy więc od przygotowania klasy, którą później będziemy testować. Poniżej przedstawiłem klasę pomagającą w zarządzaniu anonimowymi ankietami.

Plik survey.py:

```
class AnonymousSurvey:  
    """Przechowuje anonimowe odpowiedzi na pytania w ankcie."""  
  
    def __init__(self, question): ❶  
        """Przechowuje pytanie i przygotowuje do przechowywania odpowiedzi."""  
        self.question = question  
        self.responses = []  
  
    def show_question(self): ❷  
        """Wyświetla pytanie z ankity."""  
        print(self.question)  
  
    def store_response(self, new_response): ❸  
        """Przechowuje pojedynczą odpowiedź na pytanie z ankity."""  
        self.responses.append(new_response)
```

```
def show_results(self): ❸
    """Wyświetla wszystkie udzielone odpowiedzi."""
    print("Oto wyniki ankiety:")
    for response in self.responses:
        print(f"- {response}")
```

---

Na początku klasy znajduje się pytanie przeznaczone do wyświetlenia w ankiecie (patrz wiersz ❶) oraz pusta lista przeznaczona do przechowywania odpowiedzi. Klasa zawiera metody odpowiadające za wyświetlanie pytania (patrz wiersz ❷), dodanie nowej odpowiedzi do listy (patrz wiersz ❸) oraz wyświetlenie wszystkich odpowiedzi przechowywanych na liście (patrz wiersz ❹). Aby utworzyć egzemplarz na podstawie tej klasy, wystarczy, że dostarczysz pytanie zadawane w ankiecie. Kiedy zostanie utworzony egzemplarz reprezentujący daną ankietę, znajdujące się w niej pytanie będzie wyświetlane za pomocą metody `show_question()`, za przechowywanie odpowiedzi będzie odpowiedzialna metoda `store_response()`, natomiast wyniki zostaną wyświetcone przez metodę `show_results()`.

Aby pokazać, że klasa `AnonymousSurvey` działa, utworzymy teraz program wykorzystujący tę klasę.

#### Plik `language_survey.py`:

---

```
from survey import AnonymousSurvey

# Zdefiniowanie pytania i utworzenie ankiety.
question = "Jaki jest Twój ojczysty język?"
language_survey = AnonymousSurvey(question)

# Wyświetlenie pytania i przechowywanie odpowiedzi na nie.
language_survey.show_question()
print("Wpisz 'q', aby zakończyć działanie programu.\n")
while True:
    response = input("Język: ")
    if response == 'q':
        break
    language_survey.store_response(response)

# Wyświetlenie wyników ankiety.
print("\nDziękujemy każdemu respondentowi za udział w ankiecie!")
language_survey.show_results()
```

---

W powyższym programie zostało zdefiniowane pytanie „Jaki jest Twój ojczysty język?” oraz został utworzony obiekt `AnonymousSurvey` zawierający to pytanie. Program wywołuje metodę `show_question()` w celu wyświetlenia pytania, a następnie oczekuje na podanie odpowiedzi. Każda otrzymana odpowiedź zostanie zapisana. Kiedy wprowadzone zostaną już wszystkie odpowiedzi (to znaczy, kiedy użytkownik wpisał q), metoda `show_results()` wyświetli wyniki ankiety.

---

Jaki jest Twój ojczysty język?  
Wpisz 'q', aby zakończyć działanie programu.

Język: angielski  
Język: hiszpański  
Język: angielski  
Język: polski  
Język: q

Dziękujemy każdemu respondentowi za udział w ankiecie!

Oto wyniki ankiety:

- angielski
  - hiszpański
  - angielski
  - polski
- 

Przedstawiona klasa sprawdza się podczas przeprowadzania prostej, anonimowej ankiety. Założymy jednak, że chcemy usprawnić klasę `AnonymousSurvey` i moduł, w którym się ona znajduje, czyli `survey`. Można na przykład pozwolić użytkownikowi na podanie więcej niż tylko jednej odpowiedzi. Można przygotować metodę wyświetlającą jedynie unikatowe odpowiedzi i informację, ile razy poszczególne odpowiedzi zostały udzielone. Można też utworzyć inną klasę przeznaczoną do zarządzania nianonimowymi ankietami.

Implementacja wymienionych zmian wiąże się z ryzykiem uszkodzenia aktualnie działających funkcjonalności klasy `AnonymousSurvey`. Na przykład istnieje niebezpieczeństwo, że kiedy będziemy chcieli umożliwić użytkownikowi udzielenie wielu odpowiedzi, wówczas przez przypadek zmienimy sposób obsługi pojedynczych odpowiedzi. Aby mieć pewność, że nie zostaną uszkodzone istniejące funkcjonalności klasy, możemy przygotować odpowiednie testy sprawdzające jej działanie.

## Testowanie klasy `AnonymousSurvey`

Przystępujemy do utworzenia testu weryfikującego jeden z aspektów działania klasy `AnonymousSurvey`. Przygotowany test będzie sprawdzał, czy pojedyncza odpowiedź na pytanie pojawiające się w ankiecie jest właściwie przechowywana.

*Plik `test_survey.py`:*

---

```
from survey import AnonymousSurvey

def test_store_single_response(): ❶
    """Sprawdzenie, czy pojedyncza odpowiedź jest prawidłowo przechowywana."""
    question = "Jaki jest Twój ojczysty język?"
    language_survey = AnonymousSurvey(question) ❷
    language_survey.store_response('polski')
    assert 'polski' in language_survey.responses ❸
```

---

Rozpoczynamy od zimportowania klasy, która ma być przetestowana, czyli tutaj `AnonymousSurvey`. Pierwsza funkcja testowa sprawdza, czy zapis odpowiedzi na pytanie zadane w ankiecie przebiegł prawidłowo i odpowiedź ta na pewno znalazła się na liście udzielonych odpowiedzi. Dobrą opisową nazwą tej funkcji jest `test_store_single_response()` (patrz wiersz ①). Gdy test zakończy się niepowodzeniem, wówczas wyświetlona w podsumowaniu danych wyjściowych nazwa funkcji z niezaliczonym testem od razu wskaże, że problem dotyczy przechowywania pojedynczej odpowiedzi na pytanie zadane w ankiecie.

Aby przetestować zachowanie klasy, musimy utworzyć jej egzemplarz. W wierszu ② tworzymy więc egzemplarz o nazwie `language_survey` wraz z pytaniem: „Jaki jest Twój ojczysty język?”. Pojedynczą odpowiedź (`polski`) zapisujemy za pomocą metody `store_response()`. Następnie potwierdzamy prawidłowe zapisanie odpowiedzi, używając do tego asercji sprawdzającej istnienie elementu `polski` na liście `language_survey.responses` (patrz wiersz ③).

Domyślnie wydanie polecenia `pytest` bez argumentów spowoduje wykonanie wszystkich testów odkrytych przez `pytest` w katalogu bieżącym. Aby skoncentrować się tylko na testach zdefiniowanych w jednym pliku, należy nazwę tego pliku podać jako argument polecenia `pytest`. W omawianym przykładzie zostanie sprawdzony jedynie test zdefiniowany dla klasy `AnonymousSurvey`.

---

```
$ pytest test_survey.py
===== test session starts =====
--cięcie--
test_survey.py . [100%]
===== 1 passed in 0.01s =====
```

---

Wprawdzie to dobrze, ale ankieta będzie naprawdę użyteczna, jeśli pozwoli na wygenerowanie więcej niż tylko jednej odpowiedzi. Sprawdzamy więc, czy trzy odpowiedzi mogą być prawidłowo zapisane. W tym celu dodajemy następną funkcję do zestawu `TestAnonymousSurvey`:

---

```
from survey import AnonymousSurvey

def test_store_single_response():
    --cięcie--

def test_store_three_responses():
    """Sprawdzenie, czy trzy pojedyncze odpowiedzi są prawidłowo przechowywane."""
    question = "Jaki jest Twój ojczysty język?"
    language_survey = AnonymousSurvey(question)
    responses = ['angielski', 'hiszpański', 'polski'] ①
    for response in responses:
        language_survey.store_response(response)

    for response in responses: ②
        assert response in language_survey.responses
```

---

Nowej funkcji nadajemy nazwę `test_store_three_responses()`. Tworzymy tutaj obiekt ankiety, podobnie jak to zrobiliśmy w funkcji `test_store_single_response()`. Definiujemy listę zawierającą trzy różne odpowiedzi (patrz wiersz ❶), a następnie wywołujemy `store_response()` dla każdej udzielonej odpowiedzi. Po zapisaniu wszystkich odpowiedzi przechodzimy w kodzie do następnej pętli, w której wykonujemy asercję dla każdego elementu przechowywanego teraz na liście `language_survey.responses` (patrz wiersz ❷).

Kiedy ponownie uruchamiamy plik testów, oba testy (dla pojedynczej odpowiedzi oraz dla trzech odpowiedzi) zostają zaliczone:

---

```
$ pytest test_survey.py
=====
test session starts =====
--cięcie--
test_survey.py .. [100%]
===== 2 passed in 0.01s =====
```

---

Przedstawione rozwiązanie działa doskonale. Jednak kod w testach się powtarza, więc wykorzystamy tutaj następną funkcjonalność modułu `pytest`, która pozwoli poprawić efektywność testów.

## Używanie danych testowych

W programie `test_survey.py` w każdej funkcji testowej utworzyliśmy nowy egzemplarz klasy `AnonymousSurvey` oraz przygotowaliśmy nowe odpowiedzi. Takie podejście sprawdza się w niewielkim przykładzie. Jednak w rzeczywistych projektach obejmujących dziesiątki bądź setki testów będzie to problematyczne.

W testowaniu tzw. *dane testowe* umożliwiają przygotowanie środowiska testowego. Często to oznacza tworzenie zasobu używanego przez więcej niż jeden test. W module `pytest` utworzenie danych testowych odbywa się przez zdefiniowanie funkcji z dekoratorem `@pytest.fixture`. Ów *dekorator* to dyrektywa umieszczona przed definicją funkcji. Python używa tej dyrektywy przed uruchomieniem funkcji, aby zmienić sposób działania kodu funkcji. Nie przejmuj się, jeśli to brzmi zawile. Możesz rozpocząć stosowanie dekoratorów dostarczanych przez pakiety zewnętrzne, zanim nauczysz się samodzielnego tworzenia dekoratorów.

Wykorzystamy teraz dane testowe do utworzenia pojedynczego egzemplarza ankiety, który będzie mógł być używany przez obie funkcje testów zdefiniowane w pliku `test_survey.py`.

---

```
import pytest
from survey import AnonymousSurvey
```

```
@pytest.fixture ❶
def language_survey(): ❷
    """
```

Utworzenie ankiety do użycia

```

we wszystkich funkcjach testowych.

"""
question = "Jaki jest Twój ojczysty język?"
language_survey = AnonymousSurvey(question)
return language_survey

def test_store_single_response(language_survey): ❸
    """Sprawdzenie, czy pojedyncza odpowiedź jest prawidłowo przechowywana."""
    language_survey.store_response('polski') ❹
    assert 'polski' in language_survey.responses

def test_store_three_responses(language_survey): ❺
    """Sprawdzenie, czy trzy pojedyncze odpowiedzi są prawidłowo przechowywane."""
    responses = ['angielski', 'hiszpański', 'polski']
    for response in responses:
        language_survey.store_response(response) ❻

    for response in responses:
        assert response in language_survey.responses

```

---

Teraz konieczne jest zimportowanie modułu `pytest`, ponieważ używamy zdefiniowanego w nim dekoratora, `@pytest.fixture` (patrz wiersz ❶), który jest zastosowany dla nowej funkcji `language_survey()` (patrz wiersz ❷).

Zwróć uwagę na zmianę definicji obu funkcji, ❸ i ❺ — każda z nich ma teraz nowy parametr o nazwie `language_survey`. Gdy parametr funkcji testowej zostanie dopasowany do nazwy funkcji z dekoratorem `@pytest.fixture`, ta druga funkcja zostanie wykonana automatycznie, a jej wartość zwrotna zostanie przekazana funkcji testu. W omawianym przykładzie funkcja `language_survey()` dostarcza egzemplarz `language_survey` funkcjom `test_store_single_response()` i `test_store_three_responses()`.

Funkcje testów nie zawierają nowego kodu. Zwróć jednak uwagę na usunięcie dwóch wierszy w tych funkcjach, ❻ i ❾ — jeden zawierał zdefiniowane pytanie, drugi zaś odpowiadał za utworzenie obiektu `AnonymousSurvey`.

Kiedy ponownie uruchomimy plik testów, oba testy zostaną zaliczone. Tego rodzaju testy będą szczególnie użyteczne, gdy spróbujesz rozbudować klasę `AnonymousSurvey` o obsługę wielu odpowiedzi udzielanych przez każdego respondenta. Gdy zmodyfikujesz kod tak, aby umożliwiał akceptację wielu odpowiedzi, będziesz mógł ponownie przeprowadzić testy i upewnić się, że zmiany nie uszkodziły dotychczasowych funkcjonalności klasy, jeśli chodzi o obsługę pojedynczej odpowiedzi lub ich serii.

Przedstawiona tutaj struktura zdecydowanie będzie wyglądała na skomplikowaną. Zawiera bardziej abstrakcyjny kod niż zamieszczony we wcześniejszej części książki. Nie od razu musisz korzystać z danych testowych — znacznie lepiej jest tworzyć testy zawierające powtarzający się kod niż w ogóle zrezygnować z testów. Musisz tylko wiedzieć, że istnieją doskonale rozwiązania pozwalające poradzić sobie z powtarzającym się kodem testów. Ponadto w tak prostych przykładach jak tutaj omówiony dane testowe nie prowadzą do zmniejszenia ilości

kodu ani jego uproszczenia. Natomiast w projektach z wieloma testami lub w sytuacjach, w których konieczne jest użycie wielu wierszy kodu do utworzenia zasobu wykorzystywanego w wielu testach, dane testowe mogą znacznie poprawić kod przeznaczony do obsługi testów.

Kiedy chcesz przygotować dane testowe, utwórz funkcję generującą zasób używany później w wielu funkcjach testów. Do nowej funkcji dodaj dekorator `@pytest.fixture`. Następnie nazwę tej funkcji podaj jako parametr w każdej funkcji testu używającej danego zasobu. Testy staną się krótsze oraz łatwiejsze do utworzenia i późniejszej obsługi technicznej.

## ZRÓB TO SAM

**11.3. Pracownik.** Przygotuj klasę o nazwie `Employee`. Metoda `__init__()` powinna pobierać imię, nazwisko i roczne wynagrodzenie, a następnie zapisywać te informacje w postaci atrybutów. Utwórz metodę o nazwie `give_raise()`, która spowoduje zwiększenie wynagrodzenia domyślnie o 5000 zł, choć zaakceptuje także inną kwotę.

Przygotuj plik testów dla klasy `Employee`. Utwórz dwie funkcje testowe `test_give_default_raise()` i `test_give_custom_raise()`. Zaczni od utworzenia testów niewykorzystujących danych testowych i upewnij się, że są zaliczone. Następnie przygotuj dane testowe, aby uniknąć konieczności tworzenia nowego egzemplarza klasy `Employee` w każdej funkcji testowej. Ponownie wykonaj testy i upewnij się, że oba testy zostaną zaliczone.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak tworzyć testy dla funkcji i klas za pomocą narzędzi oferowanych przez moduł `pytest`. Nauczyłeś się tworzyć funkcje testowe sprawdzające określone zachowanie funkcji i klas. Zobaczyłeś, jak można wykorzystać dane testowe w celu efektywnego utworzenia zasobów, które następnie będą mogły być używane we wszystkich funkcjach testowych w danym pliku testów.

Testowanie to bardzo ważny temat, pomijany przez wielu początkujących. Nie musisz tworzyć testów dla wszystkich prostych projektów, które wypróbowujesz jako początkujący. Jednak gdy zaczniesz pracować nad projektami wymagającymi znacznie większej ilości wysiłku programistycznego, wówczas powinieneś testować wszystkie funkcje i klasy, które mają istotne znaczenie dla programu. W ten sposób będziesz miał pewność, że wprowadzone w projekcie modyfikacje nie spowodują uszkodzenia istniejącej funkcjonalności, a tym samym zyskasz swobodę we wprowadzaniu usprawnień. Jeżeli przypadkowo uszkodzisz istniejącą funkcjonalność, dzięki testom jednostkowym natychmiast się o tym dowiesz i będziesz mógł łatwo wyeliminować problem. Reakcja na niezaliczony test jest znacznie łatwiejsza niż reakcja na zgłoszenie błędu wysiane przez niezadowolonego użytkownika.

Inni programiści będą darzyć Twoje projekty większym szacunkiem, gdy umieścisz w nich przynajmniej początkowe testy jednostkowe. Dzięki temu będą odczuwać większy komfort podczas pracy z Twoim kodem i chętniej pomogą Ci w rozwijaniu projektu. Jeżeli chcesz mieć swój wkład w projektach prowadzonych przez innych programistów, będziesz musiał pokazać, że utworzony przez Ciebie kod zalicza istniejące testy. Ponadto zwykle będziesz musiał przygotować testy dla funkcjonalności, którą dodałeś do projektu.

Poeksperymentuj z testami, aby oswoić się z procesem testowania kodu. Utwórz testy dla najważniejszych komponentów funkcji i klas, ale nie staraj się zapewnić pełnego pokrycia testami wcześniejszych projektów, o ile nie masz ku temu ważnego powodu.

# Część II

## Projekty

GRATULACJE! MASZ TERAZ WYSTARCZAJĄCĄ WIEDZĘ O JĘZYKU PROGRAMOWANIA PYTHON, ABY ZACZĄĆ TWORZYĆ INTERAKTYWNE I SENSOWNE PROJEKTY. PRACA NAD WŁASNYM PROJEKTEM POZWOLI CI NA OPANOWANIE NOWYCH UMIEJĘTNOŚCI i jednocześnie pomoże w utrwaleniu koncepcji poznanych w części pierwszej książki.

W tej części książki przedstawię trzy różne rodzaje projektów, z których do realizacji możesz wybrać tylko jeden, lub nawet wszystkie trzy i wykonać je w dowolnej kolejności. Poniżej przedstawiłem krótkie omówienie poszczególnych projektów, co powinno Ci pomóc w podjęciu decyzji, którym projektem zająć się najpierw.

### Inwazja obcych, czyli utworzenie gry w Pythonie

W projekcie zatytułowanym *Inwazja obcych* (patrz rozdziały od 12. do 14.) wykorzystamy pakiet Pygame do opracowania gry 2D, w której zadaniem gracza jest zestrzelenie floty obcych pojawiających się na górze ekranu i poruszających w dół. Wykonanie zadania utrudnia zwiększące się tempo rozgrywki. W trakcie realizacji projektu nabędziesz umiejętności, które pozwolą Ci później samodzielnie tworzyć w Pythonie własne gry 2D.

# Wizualizacja danych

Projekt dotyczący wizualizacji danych rozpoczęliśmy w rozdziale 15., w którym dowiesz się, jak generować dane oraz jak utworzyć serię funkcjonalnych i pięknych wizualizacji danych za pomocą matplotlib i plotly. W rozdziale 16. pokażę, jak uzyskiwać dostęp do danych znajdujących się w internecie, a także jak dostarczać je pakietowi wizualizacji w celu utworzenia wykresu na przykład danych pogodowych lub mapy globalnej aktywności tektonicznej ziemi. Na koniec w rozdziale 17. zobaczysz, jak utworzyć program automatycznie pobierający dane i przygotowujący wizualizację na ich podstawie. Umiejętność tworzenia wizualizacji pozwoli Ci zajmować się obszarem analizy danych, co jest obecnie niezwykle poszukiwaną umiejętnością.

# Aplikacje internetowe

W trzecim projekcie (patrz rozdziały od 18. do 20.) wykorzystamy pakiet Django do utworzenia prostej aplikacji internetowej pozwalającej użytkownikom prowadzić dziennik zawierający dowolną liczbę tematów poznawanych przez nich w trakcie nauki. Użytkownicy będą mogli utworzyć konto wraz z nazwą użytkownika i hasłem, podać temat, a następnie dodawać wpisy dokumentujące proces nauki. Przy okazji omawiania tego projektu zobaczysz również, jak przebiega wdrożenie aplikacji. Dzięki temu każda osoba na świecie będzie mogła z niej korzystać.

Po ukończeniu tego projektu będziesz potrafił samodzielnie tworzyć własne, proste aplikacje internetowe. Zdobędziesz także podstawę wiedzę przydatną podczas dalszego zagłębiania się w tajniki budowania aplikacji internetowych za pomocą framework'a Django.

# 12

## Statek, który strzela pociskami



OPRACUJMY GRĘ ZATYTULOWANĄ *INWAZJA OBCYCH*! WYKORZYSTAMY W TYM CELU PYGAME, CZYLI KOLEKCJĘ ZABAWNYCH, OFERUJĄCYCH POTĘŻNE MOŻLIWOŚCI MODUŁÓW PYTHONA ODPOWIEDZIALNYCH ZA ZARZĄDZANIE GRAFIKĄ, ANIMACJĄ, A NAWET DŹWIĘKIEM. WSPOMNIAНЕ MODUŁY NIEZWYKLE UŁATWIĄĄ TWORZENIE ZAAWANSOWANYCH GIER. SKORO MODUŁY PYGAME ZAJMĄ SIĘ OBSŁUGI OPERACJI ZWIĄZANYCH Z GENEROWANIEM GRAFIKI NA EKRANIE, BĘDZIESZ MÓGL POMINAĆ WIELE ŻMUDNYCH I TRUDNYCH ZADAŃ, A ZAMIAST TEGO SKONCENTROWAĆ SIĘ NA STWORZENIU GRY O WYSOKIM POZIOMIE DYNAMIKI.

W tym rozdziale skonfigurujemy Pygame, a następnie utworzymy statek kosmiczny poruszający się w lewą i prawą stronę oraz strzelający pociskami w odpowiedzi na dane wejściowe pochodzące od użytkownika. W kolejnych dwóch rozdziałach przygotujemy flotę obcych, która będzie przeznaczona do zniszczenia, a później będziemy kontynuować dodawanie usprawnień, takich jak ograniczenie liczby statków możliwych do wykorzystania czy obsługa punktacji.

Począwszy od tego rozdziału będziesz się uczył również zarządzania dużymi projektami obejmującymi wiele plików. Przeprowadzimy refaktoryzację sporej ilości kodu, a także będziemy tak zarządzać plikami, aby projekt pozostał jak najbardziej zorganizowany, a kod działał efektywnie.

Stworzenie gry to idealny sposób na zapewnienie sobie rozrywki podczas nauki języka programowania. Ograniczoną radość przynosi obserwowanie innych osób grających w utworzoną przez nas samodzielnie grę. Zbudowanie prostej gry pomoże w zrozumieniu procesu tworzenia profesjonalnych gier. W trakcie realizacji

tego projektu wprowadzaj kod i uruchamiaj go, aby dokładnie poznać sposób działania każdego bloku kodu, który ostatecznie znajdzie się w grze. Eksperymentuj z różnymi wartościami i ustawieniami, co pozwoli Ci później jeszcze lepiej dopracować interakcje w tworzonych przez Ciebie grach.

**UWAGA** *Gra Inwazja obcych będzie składała się z wielu plików, więc w systemie utwórz dla niej nowy katalog o nazwie alien\_invasion. Aby polecenia import działały prawidłowo, upewnij się, że zapisales wszystkie pliki projektu w wymienionym katalogu.*

*Jeżeli potrafisz komfortowo pracować z systemem kontroli wersji, możesz go wykorzystać w tym projekcie. Natomiast jeśli jeszcze nie miałeś okazji używać systemu kontroli wersji, omówienie jednego z nich znajdziesz w dodatku D.*

## Planowanie projektu

Podczas pracy nad dużym projektem ważne jest przygotowanie planu przed rozpoczęciem tworzenia kodu. Dzięki planowi pozostaniesz skoncentrowany na wyznaczonych celach i będziesz miał znacznie większe szanse na ukończenie projektu.

Przygotujemy teraz ogólny opis gry. Wprawdzie ten opis nie przedstawia każdego aspektu gry *Inwazja obcych*, ale przynajmniej pokazuje, jak rozpocząć budowanie gry.

W grze *Inwazja obcych* gracz kontroluje statek kosmiczny wyświetlany na dole ekranu. Gracz może poruszać statkiem w lewą i prawą stronę za pomocą klawiszy kurSORA oraz strzelać, używając do tego spacji. Po rozpoczęciu rozgrywki na ekranie pojawia się flota obcych, którzy poruszają się wzduż ekranu oraz w kierunku statku kosmicznego gracza. Zadaniem gracza jest zestrzeliwanie obcych. Kiedy grający unicestwi wszystkich obcych, na ekranie pojawia się nowa flota przeciwników, którzy poruszają się szybciej niż wcześniej. Gdy którykolwiek obcy zetknie się ze statkiem kosmicznym lub dotrze do dolnej krawędzi ekranu, gracz traci jeden statek kosmiczny. Jeżeli straci trzy statki kosmiczne, rozgrywka zostaje zakończona.

W pierwszym etapie pracy utworzymy statek kosmiczny, który będzie mógł się poruszać w prawą i lewą stronę. Kiedy gracz naciśnie spację, statek powinien wystrzelić pocisk. Po zdefiniowaniu oczekiwanej zachowania skierujemy naszą uwagę na obcych i spróbujemy dopracować rozgrywkę.

## Instalacja Pygame

Zanim rozpocznesz tworzenie kodu, musisz zainstalować Pygame. Instalacja odbędzie się w taki sam sposób, w jaki został zainstalowany moduł `pytest` w poprzednim rozdziale. Jeżeli pominąłeś rozdział 11. lub chcesz sobie przypomnieć wiadomości na temat menedżera pakietów pip, wróć do podrozdziału „Instalowanie `pytest` za pomocą pip” w poprzednim rozdziale.

Aby zainstalować Pygame, z poziomu powłoki wydaj następujące polecenie:

---

```
$ python -m pip install --user pygame
```

---

Jeżeli do uruchamiania programów używasz polecenia innego niż `python`, np. `python3`, użyj go.

## Rozpoczęcie pracy nad projektem gry

Pracę rozpoczynamy od utworzenia pustego okna Pygame, w którym później będziemy wyświetlać elementy gry, takie jak statek kosmiczny kierowany przez gracza i pojazdy obcych. Musimy sprawić, aby gra reagowała na wprowadzane przez użytkownika dane wejściowe, a ponadto musimy zdefiniować kolor tła i wczytać obraz statku kosmicznego.

### Utworzenie okna Pygame i reagowanie na działania użytkownika

Na początek tworzymy puste okno Pygame przez zdefiniowanie klasy reprezentującej grę. W edytorze tekstu utwórz nowy plik, zapisz go pod nazwą `alien_invasion.py` i umieść w nim następujący kod:

*Plik alien\_invasion.py:*

```
import sys

import pygame

class AlienInvasion:
    """Ogólna klasa przeznaczona do zarządzania zasobami i sposobem działania
    gry."""
    def __init__(self):
        """Inicjalizacja gry i utworzenie jej zasobów."""
        pygame.init() ①

        self.screen = pygame.display.set_mode((1200, 800)) ②
        pygame.display.set_caption("Inwazja obcych")

    def run_game(self):
        """Rozpoczęcie pętli głównej gry."""
        while True: ③
            # Oczekивание на нажатие клавиши или нажатие мыши.
            for event in pygame.event.get(): ④
                if event.type == pygame.QUIT: ⑤
                    sys.exit()
```

```
# Wyświetlenie ostatnio zmodyfikowanego ekranu.  
pygame.display.flip() ❶

---



```
if __name__ == '__main__':  
    # Utworzenie egzemplarza gry i jej uruchomienie.  
    ai = AlienInvasion()  
    ai.run_game()
```


```

Na początek importujemy moduły sys i pygame. Moduł pygame zawiera funkcjonalność niezbędną do przygotowania gry. Z kolei z modułu sys skorzystamy, gdy wystąpi konieczność zakończenia gry na żądanie gracza.

Budowana tutaj gra rozpoczyna się od klasy AlienInvasion. W metodzie `__init__()` przedstawione w wierszu ❶ wywołanie `pygame.init()` inicjalizuje ustawienia tła, wymagane do prawidłowego działania Pygame. W wierszu ❷ mamy wywołanie `pygame.set_mode()` odpowiedzialne za wyświetlenie okna, w którym będziemy umieszczać wszystkie graficzne elementy gry. Argument (1200, 800) to krotka definiująca wymiary ekranu gry. Dzięki przekazaniu wymiarów do wywołania `pygame.set_mode()` tworzymy okno o szerokości 1200 pikseli i wysokości 800 pikseli. (Te wartości możesz zmienić w zależności od wielkości używanego monitora). Utworzone okno zostaje przypisane atrybutowi `self.screen` i tym samym będzie dostępne we wszystkich metodach klasy.

Obiekt przypisany atrybutowi `self.screen` jest określany mianem *powierzchni*. Wspomniana powierzchnia w Pygame to część ekranu, na której jest wyświetlany element gry. Każdy element w grze, na przykład obcy lub statek kosmiczny kierowany przez gracza, to powierzchnia. Powierzchnia zwrócona przez wywołanie `pygame.set_mode()` przedstawia cały ekran gry. Kiedy zostanie aktywowana pętla animacji gry, ta powierzchnia będzie automatycznie odświeżana w trakcie każdej iteracji pętli, więc może być aktualniana w reakcji na działania podejmowane przez użytkownika.

Gra jest kontrolowana za pomocą metody `run_game()`. Metoda ta definiuje nieustannie działającą pętlę `while` (patrz wiersz ❸) wraz z pętlą zdarzeń oraz kodem zarządzającym uaktualnieniem ekranu. Wspomniane *zdarzenie* to akcja podejmowana przez użytkownika w trakcie rozgrywki, taka jak naciśnięcie dowolnego klawisza na klawiaturze lub przesunięcie myszy. Aby program odpowiadał na zdarzenia, tworzymy tak zwaną *pętlę zdarzeń*, która *nasłuchuje* zdarzeń i podejmuje odpowiednie działanie w zależności od rodzaju przechwyconego zdarzenia. Rozpoczynająca się w wierszu ❹ pętla `for` jest w budowanej grze pętlą zdarzeń.

Aby uzyskać dostęp do zdarzeń wykrytych przez Pygame, używamy metody `pygame.event.get()` — jej wartością zwrótną jest lista zdarzeń, które wystąpiły od chwili poprzedniego wywołania metody. Dowolne zdarzenie wywołane przez klawiaturę bądź mysz spowoduje uruchomienie pętli `for`. Wewnątrz pętli tworzymy serię poleceń `if` odpowiedzialnych za wykrycie określonego zdarzenia i reakcję na nie. Na przykład kiedy gracz kliknie przycisk zamkający okno gry, zostanie wykryte zdarzenie `pygame.QUIT`, a następnie zostanie wywołane `sys.exit()` w celu zakończenia gry (patrz wiersz ❺).

Widoczne w wierszu ❶ wywołanie `pygame.display.flip()` nakazuje Pythonowi wyświetlenie ostatnio odświeżonego ekranu. W omawianym przykładzie w trakcie każdej iteracji pętli `while` mamy pusty ekran zastępujący poprzedni, więc na razie widoczny jest tylko jeden utworzony ekran. Kiedy umieścimy na ekranie pewne elementy gry, metoda `pygame.display.flip()` będzie nieustannie aktualniać ekran, aby odzwierciedlać nowe położenie elementów i ukrywać te już niepotrzebne. W ten sposób zostanie stworzona iluzja płynnego poruszania się elementów na ekranie.

Kod umieszczony na końcu pliku powoduje utworzenie egzemplarza gry i wywołanie metody `run_game()`. Metoda ta została umieszczona w bloku `if`, który gwarantuje jej wywołanie tylko w przypadku bezpośredniego uruchomienia pliku. Dlatego jeśli uruchomisz plik `alien_invasion.py`, powinieneś zobaczyć na ekranie puste okno Pygame.

## Określenie liczby klatek na sekundę

W idealnej sytuacji gra powinna działać z taką samą szybkością, *liczbą klatek na sekundę* (ang. *frame rate*), we wszystkich systemach. Kontrolowanie tej liczby w przypadku gry uruchamianej w wielu systemach jest skomplikowanym zadaniem. Pygame oferuje dość prosty sposób na wykonanie tego zadania. Utworzymy zegar i zagwarantujemy, że podczas każdej iteracji pętli głównej będzie wykonywane jego „tyknięcie”. Jeżeli przetwarzanie pętli będzie odbywało się szybciej niż zdefiniowana częstotliwość, Pygame ustali odpowiednią długość pauzy, aby zapewnić działanie gry ze stałą liczbą klatek na sekundę.

Zegar będzie zdefiniowany w metodzie `__init__()`.

Plik `alien_invasion.py`:

---

```
def __init__(self):
    """Inicjalizacja gry i utworzenie jej zasobów."""
    pygame.init()
    self.clock = pygame.time.Clock()
    --cięcie--
```

---

Po inicjalizacji pygame następuje utworzenie egzemplarza klasy `Clock` z modułu `pygame.time`. Następnie trzeba zapewnić tyknięcie zegara na końcu pętli `while` w metodzie `run_game()`.

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        --cięcie--
        pygame.display.flip()
        self.clock.tick(60)
```

---

Metoda `tick()` pobiera jeden argument: liczbę klatek na sekundę, które mają być wyświetlane w grze. W omawianym przykładzie użyłem wartości 60, więc Pygame sprawi, że pętla będzie wykonywana dokładnie 60 razy w ciągu sekundy.

**UWAGA** Zegar Pygame powinien pomóc w zapewnieniu spójnego sposobu działania gry w większości systemów. Jeżeli wynikiem jego użycia będzie mniej spójne działanie gry w Twoim systemie, wypróbuj inną wartość metody `tick()`. Natomiast jeśli nie jesteś w stanie znaleźć dobrej liczby klatek na sekundę do użycia w swoim systemie, zupełnie pomył ten zegar i dostosuj ustawienia gry, aby działała jak najlepiej w Twoim systemie.

## Zdefiniowanie koloru tła

Pygame domyślnie tworzy czarny ekran, ale to jest nudne. Teraz zmienimy więc kolor tła w wyświetlonym oknie. Odpowiednią zmianę trzeba wprowadzić na końcu metody `__init__()`.

Plik alien\_invasion.py:

---

```
def __init__(self):
    --cięcie--
    pygame.display.set_caption("Inwazja obcych")

    # Zdefiniowanie koloru tła.
    self.bg_color = (230, 230, 230) ❶

def run_game(self):
    --cięcie--
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Odświeżenie ekranu w trakcie każdej iteracji pętli.
    self.screen.fill(self.bg_color) ❷

    # Wyświetlenie ostatnio zmodyfikowanego ekranu.
    pygame.display.flip()
    self.clock.tick(60)
```

---

Kolory w Pygame są podawane w postaci wartości barw RGB, czyli powstają z połączenia koloru czerwonego, zielonego i niebieskiego. Wartość każdego składnika zawiera się w przedziale od 0 do 255. Dlatego też kolor o wartości (255, 0, 0) to czerwony, (0, 255, 0) to zielony, a (0, 0, 255) to niebieski. Wartości RGB można mieszać i otrzymać w ten sposób ponad 16 milionów kolorów. W wartości (230, 230, 230) zostały zmieszane w równym stopniu poszczególne składowe, co w efekcie spowodowało powstanie lekko szarego koloru tła. Ten kolor zostaje przypisany atrybutowi `self.bg_color` ❶.

Wypełnienie tła utworzonym kolorem następuje w wierszu ② za pomocą metody `fill()`, która działa wraz z powierzchnią i pobiera tylko jeden argument, czyli kolor.

## Utworzenie klasy ustawień

Każda nowa funkcjonalność zaimplementowana w grze z reguły oznacza również pewne nowe ustawienia. Zamiast wprowadzać je w dowolnym miejscu kodu, przygotujemy moduł o nazwie `settings` przeznaczony dla klasy `Settings` odpowiedzialnej za przechowywanie wszystkich ustawień w jednym miejscu. Takie podejście pozwala przekazać wszystkie ustawienia jako jeden obiekt, a nie wiele pojedynczych ustawień. Dzięki temu wywołania funkcji będą prostsze i łatwiejsze do zmodyfikowania, kiedy będziemy chcieli rozbudować projekt. Aby wprowadzić zmiany w grze, wystarczy, że zmienimy pewne wartości w pliku `settings.py`, zamiast szukać poszczególnych ustawień porozrzucanych w różnych plikach projektu.

W katalogu gry (`alien_invasion`) utwórz nowy plik o nazwie `settings.py` i zdefiniuj w nim klasę `Settings`, której początkową postać przedstawiłem poniżej.

*Plik settings.py:*

---

```
class Settings:
    """Klasa przeznaczona do przechowywania wszystkich ustawień gry."""

    def __init__(self):
        """Inicjalizacja ustawień gry."""
        # Ustawienia ekranu.
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

---

Aby utworzyć egzemplarz klasy `Settings` i wykorzystać go w celu uzyskania dostępu do ustawień, należy w poniższy sposób zmodyfikować plik `alien_invasion.py`.

*Plik alien\_invasion.py:*

---

```
--cięcie--
import pygame

from settings import Settings

class AlienInvasion:
    """Ogólna klasa przeznaczona do zarządzania zasobami i sposobem działania gry."""

    def __init__(self):
        """Inicjalizacja gry i utworzenie jej zasobów."""
        pygame.init()
        self.clock = pygame.time.Clock()
        self.settings = Settings() ①
```

```

    self.screen = pygame.display.set_mode( ②
        (self.settings.screen_width, self.settings.screen_height))
    pygame.display.set_caption("Inwazja obcych")

def run_game(self):
    --cięcie--
    # Odświeżenie ekranu w trakcie każdej iteracji pętli.
    self.screen.fill(self.settings.bg_color) ③

    # Wyświetlenie ostatnio zmodyfikowanego ekranu.
    pygame.display.flip()
    self.clock.tick(60)

--cięcie--

```

---

W głównym pliku programu importujemy moduł zawierający klasę `Settings`, tworzymy egzemplarz klasy `Settings` i przypisujemy go atrybutowi `self.settings` ① po wykonaniu wywołania do `pygame.init()`. Do utworzenia ekranu (patrz wiersz ②) wykorzystujemy atrybuty `screen_width` i `screen_height` atrybutu `self.settings`, za pomocą którego pobieramy także kolor tła i wypełniamy nim ekran (patrz wiersz ③).

Po uruchomieniu pliku `alien_invasion.py` nie zauważysz żadnych zmian, ponieważ jedynie przenieśliśmy ustawienia, które już były zdefiniowane w różnych miejscach kodu. Teraz można zacząć dodawać nowe elementy na ekranie.

## Dodanie obrazu statku kosmicznego

Przystępujemy teraz do dodania statku kosmicznego w budowanej grze. Aby umieścić na ekranie statek kierowany przez gracza, wczytamy odpowiedni obraz z pliku, a następnie wykorzystamy oferowaną przez Pygame metodę `blit()` do wyświetlenia pojazdu na ekranie.

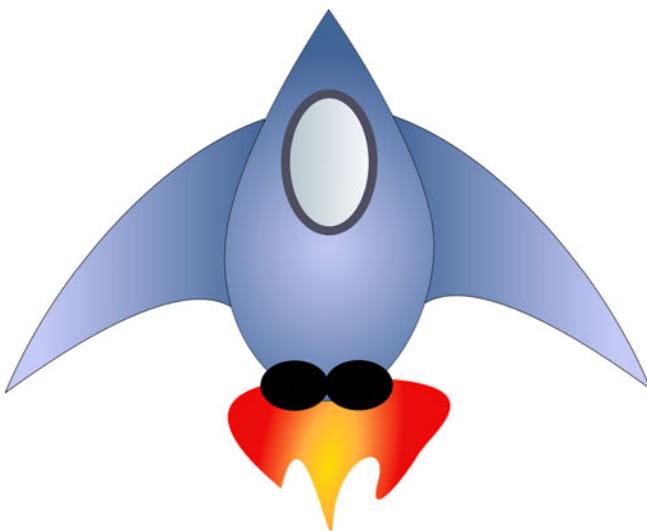
Wybierając elementy graficzne do użycia w grze, zwróć szczególną uwagę na kwestie licencji. Najbezpieczniejszym i najtańszym rozwiązaniem jest wykorzystanie bezpłatnie oferowanych elementów graficznych, które można modyfikować. Przykładem witryny oferującej tego rodzaju grafikę jest <https://opengameart.org/>.

W grze można wykorzystać praktycznie każdy rodzaj pliku graficznego, choć najłatwiej jest użyć grafiki rastrowej (pliki `.bmp`), ponieważ ten typ grafiki jest domyślnie wczytywany przez Pygame. Wprawdzie można skonfigurować Pygame tak, aby można było wykorzystywać także inne typy plików graficznych, ale niektóre z nich wymagają zainstalowania w systemie określonych bibliotek. Większość znajdowanych obrazów będzie w formacie `JPG` i `PNG`, jednak bez trudu skonwertujesz je na format `BMP` za pomocą narzędzi takich jak `Photoshop`, `GIMP` czy `Paint`.

Szczególną uwagę zwracaj na kolor tła w wybranych obrazach. Staraj się wyszukiwać takie, w których tło jest przezroczyste, ponieważ takie tło w edytorze graficznym można łatwo zastąpić dowolnym kolorem. Gra będzie prezentowała się

najlepiej, jeśli kolor tła obrazu będzie odpowiadał zdefiniowanemu kolorowi tła gry. Ewentualnie zawsze możesz dopasować kolor tła w grze do koloru obecnego na wybranym obrazie.

W budowanej grze wykorzystamy plik *ship.bmp* (patrz rysunek 12.1), który znajdziesz w materiałach przygotowanych do tej książki i dostępnych na stronie [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Kolor tła wybranego obrazu dokładnie odpowiada ustawieniom zdefiniowanym w projekcie. W katalogu głównym projektu (*alien\_invasion*) utwórz nowy podkatalog o nazwie *images*. Następnie umieść w nim plik *ship.bmp*.



Rysunek 12.1. Obraz przedstawiający statek kosmiczny, którym gracz będzie kierował w grze

## Utworzenie klasy statku kosmicznego

Skoro wybraliśmy już obraz przedstawiający statek kosmiczny, musimy go teraz wyświetlić na ekranie. Aby móc używać tego statku, przygotujemy moduł o nazwie *ship*, zawierający klasę *Ship*. Ta klasa będzie odpowiedzialna za zarządzanie praktycznie całym zachowaniem statku kosmicznego kierowanego przez gracza.

Plik *ship.py*:

---

```
import pygame

class Ship:
    """Klasa przeznaczona do zarządzania statkiem kosmicznym."""

    def __init__(self, ai_game):
        """Inicjalizacja statku kosmicznego i jego położenie początkowe."""
        # ... (remaining code)
```

```

self.screen = ai_game.screen ❶
self.screen_rect = ai_game.screen.get_rect() ❷

# Wczytanie obrazu statku kosmicznego i pobranie jego prostokąta.
self.image = pygame.image.load('images/ship.bmp') ❸
self.rect = self.image.get_rect()

# Każdy nowy statek kosmiczny pojawia się na dole ekranu.
self.rect.midbottom = self.screen_rect.midbottom ❹

def blitme(self): ❽
    """Wyświetlenie statku kosmicznego w jego aktualnym położeniu."""
    self.screen.blit(self.image, self.rect)

```

---

Jednym z powodów dużej efektywności Pygame jest możliwość traktowania elementu jako prostokąta (*rect*), nawet jeśli nie ma on dokładnego kształtu prostokąta. Efektywność związana z traktowaniem elementu jako prostokąta wynika z tego, że jest on prostą figurą geometryczną. Gdy Pygame musi ustalić, czy na przykład dwa obiekty gry kolidują ze sobą, wówczas ta operacja zostanie przeprowadzona znacznie szybciej, jeśli obiekty są prostokątami. Tego rodzaju podejście zwykle sprawdza się zaskakująco dobrze i żaden gracz nawet się nie zorientuje, że poszczególne elementy gry są przedstawiane za pomocą prostokątów. W utworzonej tutaj klasie zarówno statek, jak i ekran są traktowane jako prostokąty.

Na początku importujemy moduł `pygame`, jeszcze przed zdefiniowaniem klasy. Metoda `__init__()` klasy `Ship` pobiera dwa parametry: odniesienie do egzemplarza (`self`) oraz odniesienie do aktualnego egzemplarza klasy `AlienInvasion`. W wierszu ❶ następuje przypisanie ekranu atrybutowi egzemplarza klasy `Ship`, co daje łatwy dostęp do niego we wszystkich metodach klasy. Z kolei w wierszu ❷ za pomocą metody `get_rect()` uzyskujemy dostęp do atrybutu `rect` ekranu i przypisujemy go atrybutowi `self.screen_rect`. Dzięki temu statek kosmiczny zostaje umieszczony w odpowiednim miejscu na ekranie.

Aby wczytać obraz statku, używane jest wywołanie metody `pygame.image.load()` (patrz wiersz ❸), w którym zostało podane położenie pliku obrazu statku. Wartością zwrotną wymienionej metody jest powierzchnia przedstawiająca statek kosmiczny — przechowujemy ją w atrybucie `self.image`. Po wczytaniu obrazu używamy `get_rect()` w celu uzyskania dostępu do atrybutu `rect` powierzchni, co później pozwoli na zmianę położenia statku.

Podczas pracy z obiektem `rect` można używać współrzędnych X i Y górnej, dolnej, lewej i prawej krawędzi prostokąta, jak również jego punktu środkowego. Masz możliwość ustalenia dowolnej z tych wartości, aby ustalić aktualne położenie prostokąta. Kiedy chcesz wyśrodkować obiekt gry, do dyspozycji masz atrybuty `center`, `centerx` lub `centery` prostokąta. Z kolei podczas pracy z krawędziami ekranu możesz skorzystać z atrybutów `top`, `bottom`, `left` lub `right`. Istnieją również atrybuty łączące te właściwości, np. `midbottom`, `midtop`, `midleft` i `midright`. Do zmiany poziomego lub pionowego położenia prostokąta możesz użyć atrybutów `x` i `y`, które są współrzędnymi X i Y jego górnego lewego wierzchołka.

Wymienione atrybuty zwalniają Cię z przeprowadzania obliczeń, którymi wcześniej twórcy gier musieli zajmować się samodzielnie. Zobaczysz, że w trakcie pracy nad grą będziesz często korzystać z powyższych atrybutów.

**UWAGA** W Pygame początek, czyli punkt o współrzędnych (0, 0), znajduje się w lewym górnym rogu ekranu, natomiast wartości współrzędnych rosną w prawą stronę oraz w dół. W przypadku ekranu o rozdzielczości 1200 na 800 pikseli początek znajduje się w lewym górnym rogu ekranu, natomiast prawy dolny róg ekranu ma współrzędne (1200, 800). Te współrzędne odnoszą się do ekranu gry, a nie fizycznego ekranu monitora.

Statek kosmiczny umieszczaemy pośrodku, na dole ekranu. W tym celu upewniamy się, że wartość `self.rect.midbottom` odpowiada atrybutowi `midbottom` prostokąta przedstawiającego ekran (patrz wiersz ④). Pygame wykorzysta wymienione atrybuty prostokątów do umieszczenia obrazu statku na ekranie w taki sposób, aby był wyśrodkowany w poziomie i wyrównany do dolnej krawędzi ekranu.

W wierszu ⑤ zdefiniowaliśmy metodę `blitme()` odpowiedzialną za wyświetlenie obrazu na ekranie w położeniu wskazywanym przez `self.rect`.

## Wyświetlenie statku kosmicznego na ekranie

Przechodzimy teraz do uaktualnienia pliku `alien_invasion.py`, aby utworzyć w nim statek kosmiczny i wywoływać metodę `blitme()` tego statku.

Plik `alien_invasion.py`:

```
--cięcie--
from settings import Settings
from ship import Ship

class AlienInvasion:
    """Ogólna klasa przeznaczona do zarządzania zasobami i sposobem działania gry."""

    def __init__(self):
        --cięcie--
        pygame.display.set_caption("Inwazja obcych")

        self.ship = Ship(self) ①

    def run_game(self):
        --cięcie--
        # Odświeżenie ekranu w trakcie każdej iteracji pętli.
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme() ②

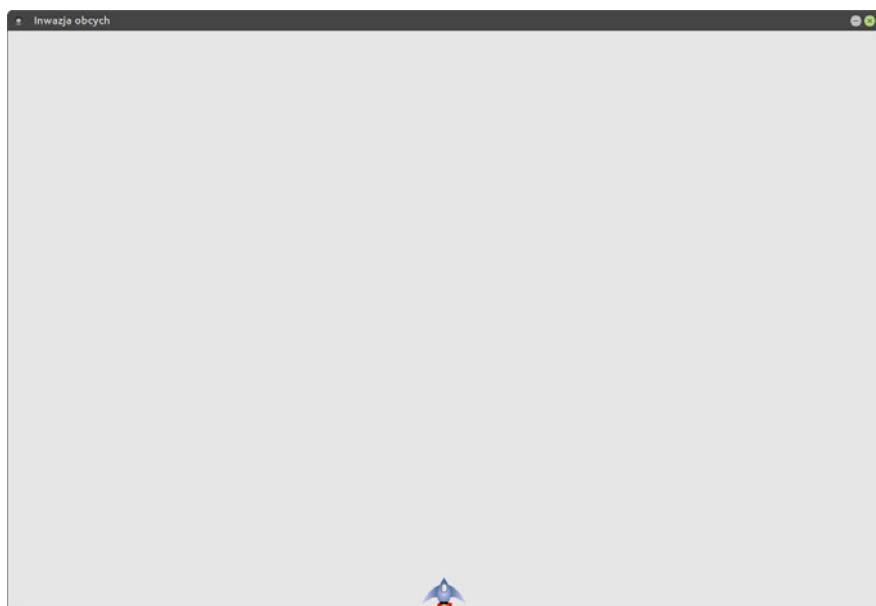
        # Wyświetlenie ostatnio zmodyfikowanego ekranu.
        pygame.display.flip()
        self.clock.tick(60)

--cięcie--
```

Importujemy moduł zawierający klasę Ship, a następnie tworzymy egzemplarz klasy Ship po utworzeniu ekranu (patrz wiersz ①). Wywołanie Ship() wymaga jednego argumentu w postaci obiektu klasy AlienInvasion. Atrybut self odwołuje się tutaj do bieżącego egzemplarza AlienInvasion. Jest to parametr zapewniający obiekowi Ship dostęp do zasobów gry takich jak obiekt screen. Wspomniany egzemplarz Ship zostaje przypisany do self.ship.

Statek wyświetlamy na ekranie za pomocą wywołania metody ship.blitme() już po zdefiniowaniu koloru tła, aby statek znajdował się nad tłem (patrz wiersz ②).

Jeżeli teraz uruchomisz program *alien\_invasion.py*, powinieneś zobaczyć pusty ekran gry wraz ze statkiem kosmicznym umieszczonym na dole ekranu, na samym środku, tak jak pokazałem na rysunku 12.2.



Rysunek 12.2. Gra „Inwazja obcych” — statek kierowany przez gracza znajduje się na dole ekranu, dokładnie pośrodku

## Refaktoryzacja, czyli metody `_check_events()` i `_update_screen()`

W większych projektach bardzo często będzie występować potrzeba refaktoryzacji utworzonego wcześniej kodu, zanim będzie można dodać kolejny. Pojęcie *refaktoryzacja* oznacza uproszczenie struktury już utworzonego kodu i ułatwienie jego dalszej rozbudowy. W tym podrozdziale coraz większa metoda `run_game()` zostanie podzielona na dwie metody pomocnicze. *Metoda pomocnicza* działa

w klasie, ale nie jest przeznaczona do wywoływania na zewnątrz klasy. W Pythonie metoda o nazwie rozpoczynającej się od jednego znaku podkreślenia wskazuje na metodę pomocniczą.

## Metoda `_check_events()`

Rozpoczynamy od przeniesienia kodu odpowiedzialnego za zarządzanie zdarzeniami do oddzielnej metody o nazwie `_check_events()`. W ten sposób uprościmy metodę `run_game()` i odizolujemy zarządzanie pętlą zdarzeń. Wspomniane odizolowanie pętli zdarzeń pozwala na zarządzanie zdarzeniami niezależnie od innych aspektów gry, takich jak uaktualnianie ekranu.

Spójrz na uaktualnioną wersję klasy `AlienInvasion` zawierającą nową metodę `_check_events()`, której wywołanie ma wpływ jedynie na kod w metodzie `run_game()`.

Plik `alien_invasion.py`:

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events() ❶

        # Odświeżenie ekranu w trakcie każdej iteracji pętli.
        --cięcie--

def _check_events(self): ❷
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

---

W przedstawionym kodzie definiujemy nową metodę `_check_events()` ❷ i przenosimy do niej polecenia sprawdzające, czy gracz kliknął przycisk zamkający okno.

W celu wywołania metody w klasie należy wykorzystać notację z użyciem kropki, zmienną `self` i nazwę metody ❶. Ta metoda jest wywoływana wewnętrz pętli `while` w metodzie `run_game()`.

## Metoda `_update_screen()`

Kod odpowiedzialny za uaktualnienie ekranu przeniesiemy do oddzielnej metody o nazwie `_update_screen()`, co pozwoli na dalsze uproszczenie metody `run_game()`.

Plik `alien_invasion.py`:

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events()
```

---

```
    self._update_screen()
    self.clock.tick(60)

def _check_events(self):
    --cięcie--

def _update_screen(self):
    """Uaktualnienie obrazów na ekranie i przejście do nowego ekranu."""
    self.screen.fill(self.settings.bg_color)
    self.ship.blitme()

    pygame.display.flip()
```

---

Do nowej metody `_update_screen()` został przeniesiony kod odpowiedzialny za wyświetlenie tła, statku i ekranu. Teraz zawartość pętli głównej w `run_game()` stała się znacznie prostsza. Bardzo łatwo można się zorientować, że w trakcie każdej iteracji pętli jest przeprowadzana obsługa nowych zdarzeń, uaktualnienie ekranu i zapewnienie odpowiedniej liczby klatek na sekundę.

Jeżeli masz już doświadczenie w tworzeniu gier, prawdopodobnie zaczniesz dzielić kod na kilka metod, podobnie jak to tutaj przedstawiłem. Natomiast jeśli nigdy wcześniej nie miałeś okazji pracować nad takimi projektami, zapewne nie wiesz, jaką strukturę powinien mieć kod. Takie podejście pokazuje, jak wygląda rzeczywisty proces tworzenia oprogramowania. Zaczynasz od przygotowania jak najprostszego kodu, a gdy projekt staje się coraz bardziej skomplikowany, przystępujesz do refaktoryzacji kodu.

Skoro przeprowadziliśmy przedstawioną w tym podrozdziale refaktoryzację, dodawanie kolejnych fragmentów kodu będzie już znacznie łatwiejsze. Możemy więc przystąpić do pracy nad dynamicznymi aspektami gry!

## ZRÓB TO SAM

**12.1. Niebieskie niebo.** Utwórz okno Pygame wraz z tłem w kolorze niebieskim.

**12.2. Postać w grze.** Wyszukaj obraz przedstawiający postać w grze oraz skonwertuj ten obraz na format rastrowy. Utwórz klasę wyświetlającą postać na środku ekranu i dopasuj kolor tła obrazu do koloru tła ekranu oraz na odwrót.

# Kierowanie statkiem kosmicznym

Umożliwimy teraz graczowi poruszanie statkiem kosmicznym w lewą i prawą stronę. W tym celu przygotujemy kod reagujący na naciśnięcie lewego i prawego klawisza kurSORA. Najpierw skoncentrujemy się na ruchu w prawą stronę, a następnie zastosujemy tę samą koncepcję do kontroli ruchu statku w lewą stronę. W trakcie wykonywania tego zadania nauczysz się kontrolować ruch obrazów na ekranie.

## Reakcja na naciśnięcie klawisza

Za każdym razem, kiedy gracz naciśnie klawisz, to naciśnięcie zostanie zarejestrowane w Pygame jako zdarzenie. Każde zdarzenie jest przechwytywane przez metodę `pygame.event.get()`, więc w przygotowanej wcześniej metodzie `_check_events()` musimy określić, jakiego rodzaju zdarzenia będziemy sprawdzać. Poszczególne naciśnięcia klawiszy są rejestrowane jako zdarzenia `KEYDOWN`.

Kiedy zostaje wykryte zdarzenie `KEYDOWN`, musimy sprawdzić, czy to naciśnięty klawisz spowodował wywołanie określonego zdarzenia. Na przykład jeśli został naciśnięty klawisz kursora w prawo, zwiększamy wartość `rect.x` statku, aby przesunąć go w prawą stronę.

*Plik alien\_invasion.py:*

```
def _check_events(self):
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN: ❶
            if event.key == pygame.K_RIGHT: ❷
                # Przesunięcie statku w prawą stronę.
                self.ship.rect.x += 1 ❸
```

Wewnątrz metody `_check_events()` do pętli zdarzeń dodaliśmy blok `elif` odpowiedzialny za właściwą reakcję po wykryciu przez Pygame zdarzenia `KEYDOWN` (patrz wiersz ❶). Sprawdzamy, czy naciśniętym klawiszem jest kurSOR w prawo (`pygame.K_RIGHT`), co odbywa się przez odczytanie atrybutu `event.key` (patrz wiersz ❷). Jeżeli został naciśnięty klawisz kursora w prawo, przesuwamy statek kosmiczny w prawą stronę przez zwiększenie wartości `self.ship.rect.x` o 1 (❸).

Jeżeli teraz uruchomisz plik `alien_invasion.py`, powinieneś zauważysz przesunięcie statku w prawą stronę po każdym naciśnięciu klawisza kursora w prawo. Wprawdzie to dopiero początek, ale już widzimy, że nie mamy do czynienia z efektywnym sposobem kontroli statku. Spróbujemy usprawnić ten aspekt gry przez wprowadzenie obsługi nieustannego ruchu.

## Umożliwienie nieustannego ruchu

Kiedy gracz naciśnie i przytrzyma klawisz kursora w prawo, chcielibyśmy, aby kierowany przez niego statek kosmiczny nieustannie przesuwał się w prawą stronę aż do chwili zwolnienia klawisza przez gracza. Musimy więc zaimplementować wykrywanie zdarzenia `pygame.KEYUP`, które poinformuje o zwolnieniu klawisza. Następnie zdarzenia `KEYDOWN` i `KEYUP` wykorzystamy razem z opcją o nazwie `moving_right`, implementując tym samym nieustanne poruszanie się statku.

Kiedy statek się nie porusza, opcja `moving_right` będzie miała wartość `False`. Gdy zostanie naciśnięty klawisz kursora w prawo, opcja otrzyma wartość `True`, a gdy klawisz zostanie zwolniony — opcja ponownie będzie miała wartość `False`.

Klasa `Ship` kontroluje wszystkie atrybuty statku kosmicznego, więc zdefiniujemy w niej atrybut o nazwie `moving_right`. Wartość tego atrybutu będziemy sprawdzać w metodzie `update()` za pomocą opcji `moving_right`. Metoda `update()` zmieni położenie statku, jeśli wartością opcji `moving_right` będzie `True`. Wymienioną metodę będziemy wywoływać za każdym razem, gdy zajdzie potrzeba uaktualnienia położenia statku.

Poniżej przedstawiłem zmiany, które należy wprowadzić w klasie `Ship`.

*Plik ship.py:*

---

```
class Ship:
    """Klasa przeznaczona do zarządzania statkiem kosmicznym."""

    def __init__(self, ai_game):
        --cięcie--
        # Każdy nowy statek kosmiczny pojawia się na dole ekranu.
        self.rect.midbottom = self.screen_rect.midbottom

        # Opcje wskazujące na poruszanie się statku.
        self.moving_right = False ①

    def update(self): ②
        """
        Uaktualnienie położenia statku na podstawie opcji wskazującej na jego ruch.
        """
        if self.moving_right:
            self.rect.x += 1

    def blitme(self):
        --cięcie--
```

---

W metodzie `__init__()` dodaliśmy atrybut `self.moving_right` oraz przypisaliśmy mu wartość początkową `False` (patrz wiersz ①). Następnie dodaliśmy metodę `update()` odpowiedzialną za poruszanie statkiem, gdy wartością wymienionej opcji jest `True` (patrz wiersz ②). Metoda `update()` jest wywoływana poprzez egzemplarz klasy `Ship`, więc nie jest uznawana za metodę pomocniczą.

Teraz zmodyfikujemy metodę `_check_events()`, aby opcji `moving_right` przypisać wartość `True` po naciśnięciu klawisza kursora w prawo oraz `False` po zwolnieniu tego klawisza.

*Plik alien\_invasion.py:*

---

```
def _check_events(self):
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        --cięcie--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True ①
```

---

```
    elif event.type == pygame.KEYUP: ❷
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = False
```

---

W wierszu ❶ modyfikujemy sposób reakcji gry na naciśnięcie przez gracza klawisza kurSORA w prawo. Zamiast bezpośrednio zmieniać położenie statku, opcji `moving_right` przypisujemy wartość `True`. W wierszu ❷ widać nowy blok `elif` odpowiadający na zdarzenia `KEYUP`. Kiedy gracz zwolni klawisz kurSORA w prawo (`K_RIGHT`), opcji `moving_right` przypiszymy wartość `False`.

Musimy jeszcze zmodyfikować pętlę `while` w metodzie `run_game()`, aby metoda `update()` klasy statku była wywoływana w trakcie każdej iteracji pętli.

*Plik alien\_invasion.py:*

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events()
        self.ship.update()
        self._update_screen()
        self.clock.tick(60)
```

---

Położenie statku zostanie uaktualnione po sprawdzeniu zdarzeń klawiatury, ale jeszcze przed uaktualnieniem ekranu. W ten sposób można zmieniać położenie statku w reakcji na dane wejściowe użytkownika oraz zagwarantować, że uaktualnione położenie zostanie odzwierciedlone podczas wyświetlania statku na ekranie.

Kiedy uruchomimy program `alien_invasion.py` i naciśniemy klawisz kurSORA w prawo, statek powinien nieustannie przesuwać się w prawą stronę aż do chwili zwolnienia klawisza.

## Poruszanie statkiem w obu kierunkach

Skoro możemy już nieustannie poruszać statkiem w prawą stronę, to obsługa ruchu w lewą stronę jest łatwa. Zadanie sprowadza się do modyfikacji klasy `Ship` oraz metody `_check_events()`. Poniżej przedstawiłem odpowiednie zmiany do wprowadzenia w metodach `__init__()` i `update()` klasy `Ship`.

*Plik ship.py:*

---

```
def __init__(self, ai_game):
    --cięcie--
    # Opcje wskazujące na poruszanie się statku
    self.moving_right = False
    self.moving_left = False

def update(self):
    """Uaktualnienie położenia statku na podstawie opcji wskazujących na jego ruch."""
    if self.moving_right:
```

```
    self.rect.x += 1
    if self.moving_left:
        self.rect.x -= 1
```

---

W metodzie `__init__()` dodaliśmy opcję `self.moving_left`. W metodzie `update()` wykorzystaliśmy dwa oddzielne bloki `if` zamiast `elif`, aby pozwolić na zwiększenie wartości `rect.x` statku, a następnie jej zmniejszenie, gdy naciśnięte są oba klawisze kurSORA. Wynikiem tego jest statek pozostający w bezruchu. Jeżeli do obsługi ruchu w lewą stronę użylibyśmy polecenia `elif`, to klawisz kurSORA w prawo zawsze miałby priorytet. Przyjęte tutaj rozwiązańe zapewnia znacznie lepsze odwzorowanie ruchu podczas przejścia od lewej do prawej strony, gdy gracz przez chwilę przytrzymuje oba klawisze kurSORA.

Konieczne jest wprowadzenie dwóch przedstawionych poniżej modyfikacji w metodzie `_check_events()`.

#### Plik alien\_invasion.py:

---

```
def _check_events(self):
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        --cięcie--
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = False
```

---

Jeżeli zdarzenie `KEYDOWN` wystąpi dla klawisza kurSORA w lewo (`K_LEFT`), opcja `moving_left` otrzyma wartość `True`. Z kolei jeśli dla klawisza `K_LEFT` wystąpi zdarzenie `KEYUP`, opcja `moving_left` będzie miała przypisaną wartość `False`. W tym miejscu możemy użyć bloków `elif`, ponieważ każde zdarzenie jest powiązane tylko z jednym klawiszem. Gdy gracz naciśnie jednocześnie dwa klawisze, wykryte zostaną dwa oddzielne zdarzenia.

Jeżeli teraz uruchomisz program `alien_invasion.py`, powinieneś mieć możliwość nieustannego poruszania statkiem kosmicznym w prawą i w lewą stronę. Po jednoczesnym naciśnięciu klawiszy kurSORA w prawo i w lewo, statek powinien się zatrzymać.

W następnej sekcji zajmiemy się dopracowaniem obsługi poruszania statkiem kosmicznym. Dostosujemy szybkość statku i ograniczmy odległość, na jaką może poruszać się statek, aby nie zniknął poza krawędzią ekranu.

## Dostosowanie szybkości statku

Aktualnie statek porusza się z szybkością jednego piksela na cykl w pętli `while`. Możemy zapewnić sobie dokładniejszą kontrolę nad szybkością poruszania się statku przez dodanie atrybutu `ship_speed` do klasy `Settings`. Ten atrybut wykorzystamy do ustalenia, jak daleko może przemieścić się statek w trakcie każdej iteracji pętli. Oto nasz nowy atrybut umieszczony w pliku `settings.py`.

*Plik settings.py:*

---

```
class Settings:
    """Klasa przeznaczona do przechowywania wszystkich ustawień gry."""

    def __init__(self):
        --cięcie--

        # Ustawienia dotyczące statku.
        self.ship_speed = 1.5
```

---

Wartość początkową atrybutu `ship_speed` zdefiniowaliśmy na 1.5. Kiedy statek będzie się poruszał, jego położenie będzie się zmieniać o 1,5 piksela zamiast tylko o 1.

Dla ustawienia dotyczącego szybkości poruszania się statku wykorzystujemy wartości zmiennoprzecinkowe, aby zachować możliwość większej kontroli nad szybkością statku, gdy zwiększy się tempo rozgrywki. Jednak atrybuty prostokąta, takie jak `x`, przechowują jedynie wartości w postaci liczb całkowitych, więc konieczne jest wprowadzenie pewnych zmian w klasie `Ship`.

*Plik ship.py:*

---

```
class Ship:
    """Klasa przeznaczona do zarządzania statkiem kosmicznym."""

    def __init__(self, ai_game):
        """Inicjalizacja statku kosmicznego i jego położenie początkowe."""
        self.screen = ai_game.screen
        self.settings = ai_game.settings ①
        --cięcie--

        # Każdy nowy statek kosmiczny pojawia się na dole ekranu.
        self.rect.midbottom = self.screen_rect.midbottom

        # Położenie poziome statku jest przechowywane w postaci liczby zmiennoprzecinkowej.
        self.x = float(self.rect.x) ②

        # Opcje wskazujące na poruszanie się statku.
        self.moving_right = False
        self.moving_left = False

    def update(self):
```

---

```

"""
Uaktualnienie położenia statku na podstawie opcji wskazujących na jego ruch.
"""

# Uaktualnienie wartości współrzędnej X statku, a nie jego prostokąta.
if self.moving_right:
    self.x += self.settings.ship_speed ❸
if self.moving_left:
    self.x -= self.settings.ship_speed

# Uaktualnienie obiektu rect na podstawie wartości self.x.
self.rect.x = self.x ❹

def blitme(self):
    --cięcie--

```

---

W wierszu ❶ tworzymy atrybut `settings` dla obiektu klasy `Ship`, aby móc go wykorzystać później w metodzie `update()`. Skoro zmieniamy położenie statku o niepełne piksele, musimy przechowywać wartość przesunięcia w zmiennej pozwalającej na obsługę liczb zmiennoprzecinkowych. Tego rodzaju wartość można przypisać atrybutowi `rect`, który jednak będzie przechowywał jedynie wartość bez części dziesiętnej. Dlatego też aby móc przechowywać dokładne położenie statku, definiujemy nowy atrybut `self.x`, który pozwala na prawidłową obsługę wartości zmiennoprzecinkowych (patrz wiersz ❷). Funkcję `float()` wykorzystaliśmy do konwersji wartości `self.rect.x` na wartość zmiennoprzecinkową, która następnie zostaje umieszczona w atrybucie `self.x`.

Teraz, kiedy w metodzie `update()` zostało zmienione położenie statku, wartość atrybutu `self.x` zostaje zmodyfikowana o wartość przechowywaną w `settings.SHIP_SPEED` (patrz wiersz ❸). Po uaktualnieniu `self.x` używamy nowej wartości do uaktualnienia wartości `self.rect.x`, która określa położenie statku (patrz wiersz ❹). W `self.rect.x` będzie przechowywana jedynie pozbawiona części ułamkowej wartość z `self.x`, ale to będzie w zupełności wystarczające do wyświetlenia statku.

Teraz dowolna wartość `SHIP_SPEED` większa niż jeden spowoduje szybsze poruszanie się statku. Dzięki takiemu rozwiązaniu statek będzie reagował wystarczająco szybko, aby było możliwe zestrzeliwanie obcych. Ponadto przyjęte podejście pozwoli na zmianę tempa rozgrywki, gdy gracz będzie czynił postępy w grze.

## Ograniczenie zasięgu poruszania się statku

Na tym etapie statek zniknie za krawędzią ekranu, gdy jeden z klawiszy kurSORA będzie naciśnięty wystarczająco dugo. Wprowadzimy teraz odpowiednie modyfikacje, aby statek przestał się poruszać po dotarciu do krawędzi ekranu. W tym celu musimy wprowadzić zmiany w metodzie `update()` w klasie `Ship`.

### Plik ship.py:

---

```
def update(self):
    """Uaktualnienie położenia statku na podstawie opcji wskazujących na jego ruch."""
    # Uaktualnienie wartości współrzędnej X statku, a nie jego prostokąta.
    if self.moving_right and self.rect.right < self.screen_rect.right: ❶
        self.x += self.settings.ship_speed
    if self.moving_left and self.rect.left > 0: ❷
        self.x -= self.settings.ship_speed

    # Uaktualnienie obiektu rect na podstawie wartości self.x.
    self.rect.x = self.x
```

---

Powyższy kod sprawdza położenie statku przed zmianą wartości `self.x`. Atrybut `self.rect.right` dostarcza wartość współrzędnej X prawej krawędzi prostokąta (`rect`) statku. Jeżeli ta wartość jest mniejsza niż przechowywana w `self.screen_rect.right`, oznacza to, że statek jeszcze nie dotarł do prawej krawędzi ekranu (patrz wiersz ❶). Taka sama sytuacja występuje w przypadku lewej krawędzi ekranu — jeżeli wartość lewej krawędzi prostokąta statku jest większa niż zero, statek nie dotarł jeszcze do lewej krawędzi ekranu (patrz wiersz ❷). W ten sposób przed zmianą wartości `self.x` sprawdzamy, czy statek kosmiczny pozostaje wciąż na ekranie.

Jeżeli teraz uruchomisz program `alien_invasion.py`, statek powinien zatrzymać się po dotarciu do krawędzi ekranu. Świetnie, niezbędna funkcjonalność została zaimplementowana za pomocą dodatkowego wyrażenia warunkowego w konstrukcji `if`. Można jednak odnieść wrażenie, że po dotarciu do krawędzi ekranu statek uderza w ścianę lub pole siłowe.

## Refaktoryzacja metody `_check_events()`

Metoda `_check_events()` będzie zawierała coraz większą ilość kodu wraz z dalszą rozbudową gry. Dlatego też warto wydzielić z niej pewien fragment kodu i umieścić go w dwóch innych metodach. Pierwsza z nowych metod będzie odpowiedzialna za obsługę zdarzeń `KEYDOWN`, natomiast druga za obsługę zdarzeń `KEYUP`.

### Plik alien\_invasion.py:

---

```
def _check_events(self):
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            self._check_keydown_events(event)
        elif event.type == pygame.KEYUP:
            self._check_keyup_events(event)

def _check_keydown_events(self, event):
    """Reakcja na naciśnięcie klawisza."""
```

```
if event.key == pygame.K_RIGHT:  
    self.ship.moving_right = True  
elif event.key == pygame.K_LEFT:  
    self.ship.moving_left = True  
  
def _check_keyup_events(self, event):  
    """Reakcja na zwolnienie klawisza."""  
    if event.key == pygame.K_RIGHT:  
        self.ship.moving_right = False  
    elif event.key == pygame.K_LEFT:  
        self.ship.moving_left = False
```

---

Utworzyliśmy dwie nowe metody pomocnicze: `_check_keydown_events()` i `_check_keyup_events()`. Obie wymagają przekazania parametrów `self` i `event`. Kod tworzący nowe metody został przeniesiony z `_check_events()`, a jego miejsce zajęły wywołania do nowych metod. W tym momencie metoda `_check_events()` stała się prostsza, ma bardziej przejrzystą strukturę kodu, co znacznie ułatwia nam dalszą jego rozbudowę i dodawanie obsługi kolejnych reakcji na dane wejściowe pochodzące od użytkownika.

## Naciśnięcie klawisza Q w celu zakończenia gry

Skoro mamy zdefiniowany efektywny mechanizm reagowania na naciśnięcia klawiszy, można dodać obsługę klawisza, którego naciśnięcie powoduje zakończenie gry. Konieczność kliknięcia myszą przycisku na górze okna gry będzie uciążliwa podczas testowania nowych funkcjonalności, więc warto dodać skrót dla tej opcji, aby koniec gry nastąpił po naciśnięciu klawisza `Q`.

Plik alien\_invasion.py:

```
def _check_keydown_events(self, event):  
    --cięcie--  
    elif event.key == pygame.K_LEFT:  
        self.ship.moving_left = True  
    elif event.key == pygame.K_q:  
        sys.exit()
```

---

W metodzie `_check_keydown_events()` został dodany nowy blok, który powoduje zakończenie gry po naciśnięciu klawisza `Q`. Teraz jeśli podczas testowania nowej funkcjonalności zechcesz zakończyć grę, wystarczy nacisnąć klawisz `Q` zamiast używać myszy do zamknięcia okna Pygame.

## Uruchamianie gry w trybie pełnoekranowym

Pygame oferuje tryb pełnoekranowy, który możesz uznać za bardziej interesujący niż uruchamianie gry w zwykłym oknie. Niektóre gry prezentują się znacznie lepiej po uruchomieniu w trybie pełnoekranowym. Ponadto użytkownicy niektórych systemów zauważą, że gra uruchomiona w trybie pełnoekranowym charakteryzuje się lepszą wydajnością działania.

Jeżeli chcesz uruchomić grę na pełnym ekranie, wprowadź przedstawione tutaj zmiany w metodzie `_init_()` zdefiniowanej w klasie `AlienInvasion`.

Plik `alien_invasion.py`:

---

```
def __init__(self):
    """Inicjalizacja gry i utworzenie jej zasobów."""
    pygame.init()
    self.settings = Settings()

    self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN) ❶
    self.settings.screen_width = self.screen.get_rect().width ❷
    self.settings.screen_height = self.screen.get_rect().height
    pygame.display.set_caption("Inwazja obcych")
```

---

Podczas tworzenia powierzchni ekranu przekazywane są wielkość w postaci `(0, 0)` i parametr `pygame.FULLSCREEN` ❶. To powoduje, że Pygame ustala taką wielkość okna, aby wypełniło ono cały ekran. Ponieważ wcześniej nie wiadomo, jakiej wielkości ekranem dysponuje użytkownik, ustawienia są korygowane po utworzeniu ekranu ❷. Atrybuty `width` i `height` prostokąta ekranu są używane do uaktualnienia obiektu `settings`.

Jeżeli lubisz wygląd lub sposób działania gier w trybie pełnoekranowym, zachowaj te ustawienia. Natomiast jeśli jesteś zwolennikiem gry w oddzielnym oknie, zawsze możesz powrócić do poprzedniego podejścia, w którym dla gry był tworzony ekran o konkretnej wielkości.

**UWAGA** *Przed uruchomieniem gry w trybie pełnoekranowym upewnij się, że będziesz mógł ją zakończyć przez naciśnięcie klawisza Q. Pygame nie oferuje domyślnego sposobu zakończenia gry uruchomionej na pełnym ekranie.*

## Krótkie powtórzenie

W kolejnym podrozdziale zajmiemy się implementacją wystrzeliwania pocisków, co będzie wymagało dodania nowego pliku o nazwie `bullet.py` oraz wprowadzenia pewnych modyfikacji w niektórych istniejących plikach. Obecnie mamy trzy pliki zawierające pewną liczbę klas i metod. Aby mieć pełne rozeznanie, jeśli chodzi o organizację projektu, przejrzymy poszczególne pliki, zanim przystąpimy do dodania następnych funkcjonalności.

### `alien_invasion.py`

Główny plik gry `alien_invasion.py` zawiera klasę `AlienInvasion`. Ta klasa tworzy pewną liczbę ważnych atrybutów używanych w trakcie gry. Tak więc ustawienia są przechowywane w `settings`, główna wyświetlana warstwa jest przechowywana w `screen`, a egzemplarz statku (`ship`) również jest tworzony przez kod w omawianym

pliku. Ten plik zawiera także pętlę główną gry (`while`) wywołującą metody `_check_events()`, `ship.update()` i `_update_screen()`. Ten moduł zapewnia również obsługę w trakcie każdej iteracji pętli głównej zegara zapewniającego odpowiednią liczbę klatek na sekundę.

Metoda `_check_events()` wykrywa zdarzenia, takie jak naciśnięcie lub zwolnienie klawisza, i przetwarza je za pomocą metod `_check_keydown_events()` i `_check_keyup_events()`. Obecnie te metody są odpowiedzialne za zarządzanie ruchem statku kosmicznego. Klasa `AlienInvasion` zawiera również metodę `_update_screen()`, aktualizującą w trakcie każdej iteracji pętli głównej elementy wyświetlane na ekranie.

Plik `alien_invasion.py` to jedyny, który trzeba uruchomić, aby móc zagrać w grę *Inwazja obcych*. Pozostałe pliki, czyli `settings.py` i `ship.py`, zawierają kod, który jest bezpośrednio lub pośrednio importowany do głównego pliku gry.

## settings.py

Omawiany plik zawiera definicję klasy `Settings`. Ta klasa ma jedynie metodę `_init_()` odpowiedzialną za inicjalizację atrybutów określających wygląd gry oraz szybkość poruszania się statku.

## ship.py

Omawiany plik zawiera definicję klasy `Ship`. W tej klasie mamy metody `_init_()`, `update()`, która jest odpowiedzialna za zarządzenie położeniem statku, oraz `blitme()`, która zajmuje się wyświetlaniem statku na ekranie. Rzeczywisty obraz statku jest przechowywany w pliku `ship.bmp`, który znajduje się w katalogu `images`.

### ZRÓB TO SAM

**12.3. Dokumentacja Pygame.** Gra na jej obecnym poziomie jest na tyle zaawansowana, że być może zechcesz zajrzeć do dokumentacji Pygame. Strona główna Pygame znajduje się pod adresem <https://www.pygame.org/>, natomiast dokumentacja jest dostępna na stronie <https://www.pygame.org/docs/>. W tym momencie wystarczy, że przejrzysz dostępną dokumentację. Wprawdzie nie musisz z niej korzystać, aby dokończyć projekt gry, ale lektura dokumentacji zdecydowanie pomoże Ci w modyfikowaniu gry *Inwazja obcych*, a także podczas samodzielnego tworzenia własnych gier.

**12.4. Rakieta.** Utwórz grę rozpoczęającą się od wyświetlenia rakiety na środku ekranu. Pozwól graczowi na poruszanie rakietą w górę, w dół, w lewo i w prawo za pomocą klawiszy kurSORA. Upewnij się, że rakieta nigdy nie przekroczy krawędzi ekranu.

**12.5. Klawisze.** Utwórz plik Pygame definiujący pusty ekran. W pętli zdarzeń wyświetl atrybut `event.key` za każdym razem, gdy zostanie wykryte zdarzenie `pygame.KEYDOWN`. Uruchom program i naciskaj różne klawisze, aby zobaczyć, w jaki sposób Pygame reaguje na zdarzenia.

# Wystrzeliwanie pocisków

Przechodzimy teraz do implementacji funkcjonalności wystrzeliwania pocisków. Przygotujemy kod pozwalający na wystrzeliwanie pocisku (mały prostokąt) po naciśnięciu klawisza spacji przez gracza. Następnie pocisk będzie poruszał się w górę ekranu i zniknął za jego górną krawędzią.

## Dodawanie ustawień dotyczących pocisków

Pracę musimy zacząć od uaktualnienia pliku *settings.py*, aby zawierał różne wartości niezbędne dla nowej klasy *Bullet*. Te wartości umieść na końcu metody *\_\_init\_\_()*.

Plik *settings.py*:

---

```
def __init__(self):
    --cięcie--
    # Ustawienia dotyczące pocisku.
    self.bullet_speed = 2.0
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = (60, 60, 60)
```

---

Powyższe ustawienia powodują utworzenie ciemnoszarego pocisku o szerokości 3 pikseli i wysokości 15 pikseli. Wystrzelony pocisk będzie poruszał się nieco szybciej niż statek kosmiczny kierowany przez gracza.

## Utworzenie klasy Bullet

Przechodzimy teraz do utworzenia pliku *bullet.py*, w którym będzie zdefiniowana klasa *Bullet*. Poniżej przedstawiłem pierwszą część kodu przeznaczonego do umieszczenia w omawianym pliku.

Plik *bullet.py*:

---

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """Klasa przeznaczona do zarządzania pociskami wystrzeliwanymi przez statek."""

    def __init__(self, ai_game):
        """Utworzenie obiektu pocisku w aktualnym położeniu statku."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color
```

---

```
# Utworzenie prostokąta pocisku w punkcie (0, 0), a następnie
# zdefiniowanie dla niego odpowiedniego położenia.
    self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
                           self.settings.bullet_height)
    self.rect.midtop = ai_game.ship.rect.midtop ❶

# Położenie pocisku jest zdefiniowane za pomocą wartości zmiennoprzecinkowej.
    self.y = float(self.rect.y) ❷

❸
```

---

Klasa Bullet dziedziczy po klasie Sprite, która została zimportowana z modułu `pygame.sprite`. Kiedy używasz sprite'ów, możesz grupować powiązane ze sobą elementy gry i przeprowadzać operacje na całej grupie elementów. Aby móc utworzyć egzemplarz pocisku, metoda `__init__()` wymaga przekazania jej aktualnego egzemplarza klasy AlienInvasion. Konieczne jest wywołanie `super()`, aby zapewnić prawidłowe dziedziczenie po klasie Sprite. Definiujemy również atrybuty przeznaczone dla obiektów ekranu i ustawień, a także koloru pocisku.

W wierszu ❶ tworzymy atrybut `rect` pocisku. Do przedstawienia pocisku nie wykorzystujemy tym razem pliku graficznego, więc konieczne jest zbudowanie prostokąta zupełnie od początku za pomocą wywołania `pygame.Rect()`. Klasa Rect wymaga podania współrzędnych X i Y lewego górnego wierzchołka prostokąta oraz jego szerokości i wysokości. Prostokąt inicjalizujemy w punkcie (0, 0), ale w kolejnych dwóch wierszach kodu przenosimy go w odpowiednie miejsce, ponieważ położenie pocisku zależy od położenia statku. Z kolei szerokość i wysokość pocisku pobieramy z wartości przechowywanych w `self.settings`.

W wierszu ❷ atrybutowi `midtop` pocisku przypisujemy te same wartości, jakie ma atrybut `midtop` statku. Pocisk powinien pojawić się na górze statku, więc górną krawędź prostokąta pocisku dopasowujemy do górnej krawędzi prostokąta statku. W ten sposób powstaje wrażenie wystrzeliwania pocisku ze statku kosmicznego. Współrzędną Y położenia pocisku przechowujemy w postaci wartości zmiennoprzecinkowej, aby zachować dokładną kontrolę nad szybkością pocisku (patrz wiersz ❸).

Poniżej przedstawiłem drugą część kodu przeznaczonego do umieszczenia w pliku `bullet.py`. Tym razem to metody `update()` i `draw_bullet()`.

*Plik bullet.py:*

---

```
def update(self):
    """Poruszanie pociskiem po ekranie."""
    # Uaktualnienie położenia pocisku.
    self.y -= self.settings.bullet_speed ❶
    # Uaktualnienie położenia prostokąta pocisku.
    self.rect.y = self.y ❷

def draw_bullet(self):
    """Wyświetlenie pocisku na ekranie."""
    pygame.draw.rect(self.screen, self.color, self.rect) ❸
```

---

Metoda `update()` zarządza położeniem pocisku. Kiedy pocisk zostaje wystrzelony, porusza się w górę ekranu, co odpowiada zmniejszeniu wartości współrzędnej Y. Tak więc w celu uaktualnienia położenia pocisku musimy odjąć wartość przechowywaną w `settings.bullet_speed` od `self.y` (patrz wiersz ①). Następnie używamy wartości `self.y` do ustawienia wartości `self.rect.y` (patrz wiersz ②).

Atrybut `bullet_speed` pozwala na zwiększenie szybkości pocisku, gdy gracz po czyni postępy w grze lub gdy zajdzie konieczność modyfikacji zachowania gry. Gdy pocisk zostanie wystrzelony, wartość jego współrzędnej X nie będzie się zmieniać, ponieważ porusza się on pionowo w górę, po linii prostej, nawet jeśli statek będzie się poruszał.

Kiedy chcemy wyświetlić pocisk, wywołujemy metodę `draw_bullet()`. Funkcja `draw.rect()` wypełnia kolorem, zdefiniowanym w atrybutie `self.color`, fragment ekranu wskazany przez prostokąt pocisku (`rect`), jak można zobaczyć w wierszu ③.

## Przechowywanie pocisków w grupie

Skoro mamy zdefiniowaną klasę `Bullet` i niezbędne ustawienia, możemy przystąpić do przygotowania kodu powodującego wystrzelanie pocisku po każdym naciśnięciu spacji przez gracza. Zaczynamy od zdefiniowania grupy w klasie `AlienInvasion` przeznaczonej do przechowywania wszystkich aktualnych pocisków, co pozwoli nam na zarządzanie wystrzelonymi pociskami. Ta grupa będzie egzemplarzem `pygame.sprite.Group`, który zachowuje się jak lista, choć oferuje pewne funkcje dodatkowe, przydatne podczas tworzenia gier. Utworzoną tutaj grupę wykorzystamy do wyświetlania pocisków na ekranie w trakcie każdej iteracji pętli głównej oraz do uaktualniania położenia poszczególnych pocisków.

Na początek trzeba zaimportować nową klasę `Bullet`.

Plik `alien_invasion.py`:

```
--cięcie--  
from ship import Ship  
from bullet import Bullet
```

Następnie w metodzie `__init__()` zostaje utworzona grupa przeznaczona do przechowywania pocisków.

Plik `alien_invasion.py`:

```
def __init__(self):  
    --cięcie--  
    self.ship = Ship(self)  
    self.bullets = pygame.sprite.Group()
```

Położenie pocisków musi być uaktualniane podczas każdej iteracji pętli `while`:

### *Plik alien\_invasion.py:*

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events()
        self.ship.update()
        self.bullets.update()
        self._update_screen()
        self.clock.tick(60)
```

---

Kiedy wywoływana jest metoda `update()` dla grupy, automatycznie jest też wywoływana metoda `update()` dla każdego sprite'a w grupie. Wiersz `self.bullets.update()` powoduje wywołanie `bullets.update()` dla każdego pocisku umieszczonego w grupie `bullets`.

## **Wystrzeliwanie pocisków**

W klasie `AlienInvasion` konieczne jest zmodyfikowanie metody `_check_keydown_events()`, aby pocisk był wystrzeliwany po każdym naciśnięciu klawisza spacji. Nie musimy zmieniać metody `_check_keyup_events()`, ponieważ nic się nie dzieje po zwolnieniu klawisza. Konieczne jest zmodyfikowanie metody `_update_screen()`, aby mieć pewność, że każdy pocisk zostanie wyświetlony na ekranie przed wywołaniem `flip()`.

Wiadomo, że po wystrzeleniu pocisku trzeba będzie wykonać pewne zadania, więc przystępujemy do zdefiniowania metody, `_fire_bullet()`, która będzie się tym zajmowała.

### *Plik alien\_invasion.py:*

---

```
def _check_keydown_events(self, event):
    --cięcie--
    elif event.key == pygame.K_q:
        sys.exit()
    elif event.key == pygame.K_SPACE: ❶
        self._fire_bullet()

def _check_keyup_events(self, event):
    --cięcie--

def _fire_bullet(self):
    """Utworzenie nowego pocisku i dodanie go do grupy pocisków."""
    new_bullet = Bullet(self) ❷
    self.bullets.add(new_bullet) ❸

def _update_screen(self):
    """Uaktualnienie obrazów na ekranie i wyświetlenie tego ekranu."""
    self.screen.fill(self.settings.bg_color)
    for bullet in self.bullets.sprites(): ❹
```

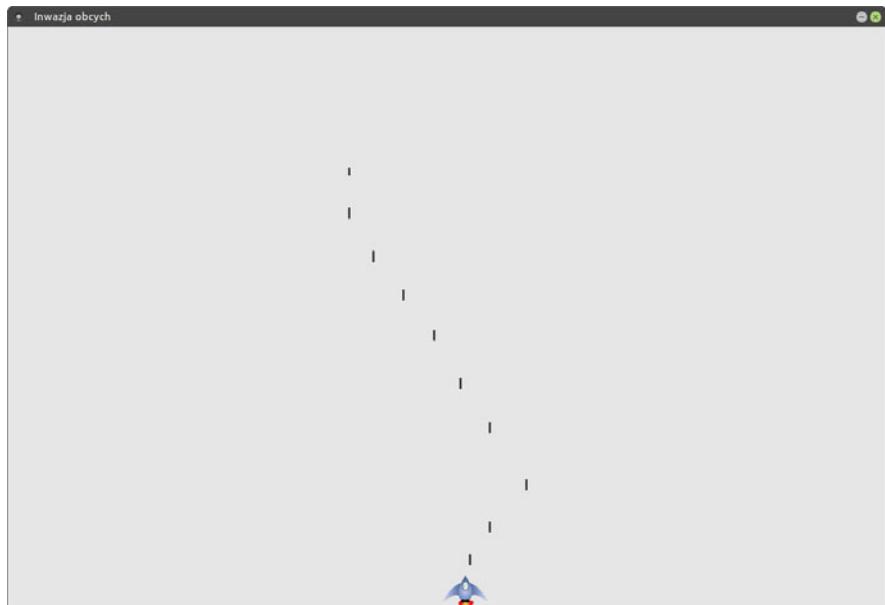
```
    bullet.draw_bullet()
self.ship.blitme()

pygame.display.flip()
--cięcie--
```

Po naciśnięciu spacji wywoływana jest metoda `_fire_bullet()` (zobacz wiersz ❶). W tej metodzie tworzymy nowy egzemplarz klasy Bullet, któremu nadajemy nazwę `new_bullet` (patrz wiersz ❷) i dodajemy go do grupy `bullets` za pomocą metody `add()`, jak pokazałem w wierszu ❸. Metoda `add()` jest podobna do `append()`, ale została utworzona specjalnie dla grup Pygame.

Wartością zwracaną `bullets.sprites()` jest lista wszystkich sprite'ów znajdujących się w grupie `bullets`. Aby wyświetlić na ekranie wszystkie wystrzelone pociski, przeprowadzamy iterację przez sprite'y w grupie `bullets` i dla każdego z nich wywołujemy metodę `draw_bullet()`, tak jak to pokazałem w wierszu ❹. Ta pętla zostaje umieszczona przed polecienniem wyświetlającym statek, aby pocisk na początku nie był wyświetlany na statku.

Jeżeli teraz uruchomisz program `alien_invasion.py`, powinieneś mieć możliwość poruszania statkiem kosmicznym w prawą i w lewą stronę, a także wystrzeliwania pocisków. Te pociski poruszają się w górę ekranu i znikają po dotarciu do górnej krawędzi (patrz rysunek 12.3). Wielkość, kolor i szybkość poruszania się pocisków możesz zmienić w pliku `settings.py`.



Rysunek 12.3. Statek kosmiczny po wystrzeleniu serii pocisków

## Usuwanie niewidocznych pocisków

Na obecnym etapie pracy nad grą pociski znikają po dotarciu do górnej krawędzi ekranu, ale tylko dlatego, że Pygame nie może ich narysować powyżej krawędzi ekranu. Tak naprawdę pociski nadal istnieją, a ich współrzędne Y po prostu mają coraz większą wartość ujemną. To stanowi problem, ponieważ pociski wciąż zajmują pewną ilość miejsca w pamięci i zabierają moc procesora wymaganą do ich dalszej obsługi.

Musimy więc pozbyć się pocisków niewidocznych na ekranie. W przeciwnym razie wydajność gry znacznie spadnie ze względu na konieczność wykonywania niepotrzebnej pracy. Zaimplementujemy zatem rozwiązanie polegające na wykryciu, kiedy wartość bottom prostokąta pocisku (rect) będzie wynosiła 0, co oznacza zniknięcie pocisku poza górną krawędzią ekranu.

Plik alien\_invasion.py:

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events()
        self.ship.update()
        self.bullets.update()

        # Usunięcie pocisków, które znajdują się poza ekranem.
        for bullet in self.bullets.copy(): ❶
            if bullet.rect.bottom <= 0: ❷
                self.bullets.remove(bullet) ❸
        print(len(self.bullets)) ❹

        self._update_screen()
        self.clock.tick(60)
```

---

Gdy pętla for jest używana z listą (lub grupą w Pygame), Python oczekuje, że wielkość listy pozostanie niezmieniona w trakcie wszystkich iteracji tej pętli. Skoro wewnętrz pętli for nie należy usuwać elementów listy lub grupy, stąd konieczność przeprowadzenia iteracji przez kopię grupy. Wykorzystujemy metodę copy() w celu przygotowania pętli for (patrz wiersz ❶) pozwalającej zmodyfikować grupę bullets wewnętrz pętli. W wierszu ❷ sprawdzamy każdy pocisk i ustalamy, czy zniknął z ekranu. Jeżeli tak, dany pocisk zostaje usunięty z grupy bullets (patrz wiersz ❸). W wierszu ❹ dodajemy nowe wywołanie print() wyświetlające liczbę aktualnie istniejących pocisków w grze, co pozwala nam uzyskać potwierdzenie, że pociski, które znikają poza górną krawędzią ekranu, są usuwane.

Jeżeli kod działa prawidłowo, podczas wystrzeliwania pocisków można obserwować dane wyjściowe w terminalu. Zobaczysz wówczas, jak liczba pocisków spada do zera, gdy wszystkie pociski przekroczą górną krawędź ekranu. Po uruchomieniu gry i potwierdzeniu prawidłowego usuwania pocisków z grupy możesz

już usunąć wspomniane wywołanie `print()`. Jeżeli je pozostawisz, wydajność gry znacznie się obniży, ponieważ wyświetlenie danych wyjściowych w terminalu zabiera znacznie więcej czasu niż wyświetlenie elementu graficznego w oknie gry.

## Ograniczenie liczby pocisków

W wielu tego rodzaju grach autorzy, aby zachętać gracza do większej celności, ograniczają liczbę pocisków, jakie mogą znajdować się na ekranie. Dokładnie takie samo rozwiązanie zastosujemy w budowanej tutaj grze *Inwazja obcych*.

Przede wszystkim definiujemy w pliku `settings.py` liczbę pocisków, które jednocześnie mogą być wyświetlane na ekranie.

*Plik settings.py:*

---

```
# Ustawienia dotyczące pocisku.  
--cięcie--  
self.bullet_color = (60, 60, 60)  
self.bullets_allowed = 3
```

---

W ten sposób gracz ma do dyspozycji jedynie trzy pociski. Zanim zostanie utworzony nowy pocisk w metodzie `_fire_bullet()`, wspomniane ustawienie wykorzystamy w klasie `AlienInvasion`, aby sprawdzić, ile pocisków znajduje się na ekranie.

*Plik alien\_invasion.py:*

---

```
def _fire_bullet(self):  
    """Utworzenie nowego pocisku i dodanie go do grupy pocisków."""  
    if len(self.bullets) < self.settings.bullets_allowed:  
        new_bullet = Bullet(self)  
        self.bullets.add(new_bullet)
```

---

Po naciśnięciu spacji sprawdzamy liczbę elementów grupy `bullets`. Jeżeli wartość zwrotna wywołania `len(self.bullets)` wynosi mniej niż trzy, tworzymy nowy pocisk. Natomiast jeśli na ekranie znajdują się trzy pociski, naciśnięcie spacji nie da żadnego efektu. Jeżeli teraz uruchomisz grę, powinieneś zobaczyć, że na ekranie mogą znajdować się co najwyżej trzy pociski.

## Utworzenie metody `_update_bullets()`

Chcemy zachować maksymalną prostotę klasy `AlienInvasion`. Dlatego też skoro przygotowaliśmy i potwierdziliśmy prawidłowe działanie kodu odpowiedzialnego za zarządzanie pociskami, możemy go przenieść do oddzielnej metody. Utworzmy nową nową metodę o nazwie `_update_bullets()`, którą umieścimy przed `_update_screen()`.

*Plik alien\_invasion.py:*

---

```
def _update_bullets(self):
    """Uaktualnienie położenia pocisków i usunięcie tych niewidocznych na ekranie."""
    # Uaktualnienie położenia pocisków.
    self.bullets.update()

    # Usunięcie pocisków, które znajdują się poza ekranem.
    for bullet in self.bullets.copy():
        if bullet.rect.bottom <= 0:
            self.bullets.remove(bullet)
```

---

Kod metody `_update_bullets()` został skopiowany z `run_game()`. Jedyna zmiana polega na doprecyzowaniu komentarza.

Pętla `while` w `run_game()` ponownie stała się prosta.

*Plik alien\_invasion.py:*

---

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update_screen()
    self.clock.tick(60)
```

---

Teraz pętla główna zawiera jedynie minimalną ilość kodu. Dzięki temu można szybko odczytać nazwy metod i ustalić, co tak naprawdę dzieje się w grze. Pętla główna sprawdza dane wejściowe pobrane od użytkownika, a następnie uaktualnia położenie statku oraz ustala liczbę pocisków znajdujących się na ekranie. Uaktualnione położenie statku i pocisków wykorzystujemy do odświeżenia ekranu.

Uruchom program `alien_invasion.py` raz jeszcze i upewnij się, że nadal masz możliwość wystrzeliwania pocisków.

### ZRÓB TO SAM

**12.6. Strzelanie na boki.** Utwórz grę, w której statek kosmiczny jest umieszczony po lewej stronie ekranu i pozwala graczowi na poruszanie się w górę oraz w dół. Po naciśnięciu klawisza spacji statek powinien mieć możliwość wystrzeliwania pocisków poruszających się w prawą stronę. Upewnij się, że pociski, które znikają z ekranu, są usuwane.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak przygotować plan gry. Poznałeś podstawową strukturę gry utworzonej w Pygame. Nauczyłeś się definiować kolor tła oraz przechowywać ustawienia w oddzielnej klasie, dzięki której są one dostępne dla wszystkich komponentów gry i można je łatwo zmieniać. Zobaczyłeś, jak wyświetlić obraz na ekranie i umożliwić graczowi sterowanie elementami wyświetlonymi na ekranie. Dowiedziałeś się, jak utworzyć elementy poruszające się samodzielnie, takie jak pociski, a także jak usuwać obiekty, które są już niepotrzebne. Nauczyłeś się przeprowadzać regularnie refaktoryzację kodu w projekcie, aby ułatwiać jego dalszą rozbudowę.

W rozdziale 13. dodamy obcych do budowanej gry. Kiedy skończysz lekturę następnego rozdziału, będziesz mógł zestrzeliwać obcych — mam nadzieję, że zanim dotrą do kierowanego przez Ciebie statku kosmicznego!

# 13

## Obcy!



W TYM ROZDZIALE ZAJMIEMY SIĘ DODAWANIEM OBCYCH DO BUDOWANEJ GRY. NAJPIERW UMIEŚCIMY JEDNEGO OBCEGO W POBLIŻU GÓRNEJ KRAWĘDZI EKRANU, A NASTĘPNIE WYGENERUJEMY CAŁĄ FLOTĘ OBCYCH.

Ta flota obcych będzie poruszać się na boki oraz do dołu, a zadanie gracza będzie polegało na pozbiciu się obcych przez ich zestrzelenie. Na końcu ograniczymy liczbę statków, jakie pozostają do dyspozycji gracza. Zakończenie gry nastąpi, gdy gracz straci ostatni przysługujący mu statek.

Podczas lektury rozdziału dowiesz się więcej na temat Pygame i zarządzania dużym projektem. Ponadto zobaczysz, jak wykrywać kolizje między obiektami gry, takimi jak pociski i pojazdy obcych. Wykrywanie kolizji między obiektami gry pomaga w zdefiniowaniu interakcji zachodzących między elementami w grze — postać możesz uwieźć w labiryncie, lub też pozwolić dwóm postaciom na przekazywanie sobie piłki. Będziemy kontynuować pracę na podstawie planu, do którego co pewien czas będziemy wracać, aby zachować koncentrację na sesjach tworzenia kodu.

Jednak zanim przystąpimy do tworzenia nowego kodu, który pozwoli nam umieścić flotę obcych na ekranie, spojrzymy na projekt i nieco uaktualnimy przygotowany wcześniej plan.

# Przegląd projektu

Kiedy w dużym projekcie zaczynasz nowy etap programowania, zawsze dobrym pomysłem jest powrót do pierwotnego planu i doprecyzowanie tego, co chcesz osiągnąć za pomocą nowego kodu. W tym rozdziale zajmiemy się następującymi zadaniami:

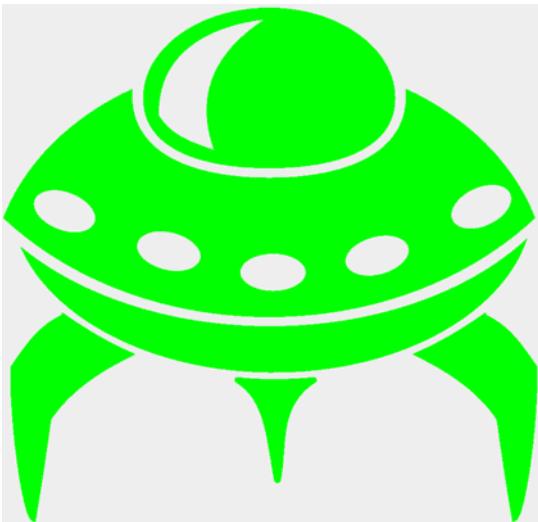
- Dodanie pojedynczego obcego w lewym górnym rogu ekranu wraz z odpowiednią ilością miejsca wokół niego.
- Na podstawie ilości wolnego miejsca obok dodanego obcego oraz ogólnej wielkości ekranu określenie liczby obcych, którzy zmieszcza się w pierwszym rzędzie. Następnie utworzenie kolejnych rzędów obcych, aby powstała ich cała flota.
- Zdefiniowanie zachowania floty obcych — porusza się na boki oraz do dołu aż do chwili zestrzelenia wszystkich obcych przez gracza, zetknięcia się jednego z obcych ze statkiem kosmicznym kierowanym przez gracza lub dotarcia jednego z obcych do dolnej krawędzi ekranu. W pierwszym przypadku zostanie utworzona nowa flota obcych, natomiast w dwóch pozostałych zostanie zniszczony statek, a następnie utworzona nowa flota obcych.
- Ograniczenie liczby statków, którymi dysponuje gracz. Gdy gracz wykorzysta wszystkie statki, wówczas gra się zakończy.

Wprawdzie powyższy plan będziemy dopracowywać podczas implementacji poszczególnych funkcjonalności, ale na początek w zupełności wystarczy.

Powinieneś również przeanalizować istniejący kod projektu, gdy przystępujesz do pracy nad dodawaniem nowej funkcjonalności. Ponieważ każda nowa faza zwykle oznacza zwiększenie poziomu skomplikowania projektu, najlepszym rozwiązaniem jest pozbycie się niepotrzebnego lub nieefektywnie działającego kodu. W budowanej grze nie mamy zbyt wiele do usunięcia, ponieważ refaktoryzację przeprowadzaliśmy na bieżąco.

## Utworzenie pierwszego obcego

Umieszczenie jednego obcego na ekranie przypomina umieszczenie na ekranie statku. Zachowanie poszczególnych obcych jest kontrolowane przez klasę o nazwie `Alien`, której nadamy strukturę podobną do tej użytej w klasie `Ship`. W celu zachowania prostoty nadal będziemy wykorzystywać obrazy rastrowe. Możesz samodzielnie wyszukać odpowiadający Ci obraz obcego lub użyć tego pokazanego na rysunku 13.1 — znajdziesz go w materiałach przygotowanych do tej książki i dostępnych na stronie [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Ten obraz ma szare tło w kolorze odpowiadającym kolorowi tła gry. Upewnij się, że wybrany przez Ciebie obraz został zapisany w katalogu `images` projektu.



Rysunek 13.1. Obcy, którego wykorzystamy do utworzenia floty

## Utworzenie klasy Alien

Przystępujemy do zdefiniowania klasy Alien. Umieść ją w pliku *alien.py*.

Plik alien.py:

---

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """Klasa przedstawiająca pojedynczego obcego we flocie."""

    def __init__(self, ai_game):
        """Inicjalizacja obcego i zdefiniowanie jego położenia początkowego."""
        super().__init__()
        self.screen = ai_game.screen

        # Wczytanie obrazu obcego i zdefiniowanie jego atrybutu rect.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Umieszczenie nowego obcego w pobliżu lewego górnego rogu ekranu.
        self.rect.x = self.rect.width ❶
        self.rect.y = self.rect.height

        # Przechowywanie dokładnego poziomego położenia obcego.
        self.x = float(self.rect.x) ❷
```

---

Większość kodu w powyższej klasie jest taka sama jak w klasie `Ship` — z wyjątkiem miejsca położenia obcego. Początkowo umieszczamy go w pobliżu lewego górnego rogu ekranu. Po lewej stronie obcego pozostawiamy ilość miejsca równą szerokości obcego, natomiast nad nim ilość miejsca równą jego wysokości (patrz wiersz ❶). Koncentrujemy się przede wszystkim na szybkości poruszania się obcego w poziomie i dlatego konieczne jest dokładne śledzenie jego położenia poziomego, jak pokazałem w wierszu ❷.

Klasa `Alien` nie wymaga metody powodującej wyświetlenie obcego na ekranie. Zamiast niej wykorzystamy metodę grupy Pygame w celu automatycznego wyświetlenia wszystkich elementów na ekranie.

## Utworzenie egzemplarza obcego

Chcemy utworzyć egzemplarz klasy `Alien`, aby można było zobaczyć pierwszego obcego na ekranie. Ponieważ to zadanie należy do programisty, kod tworzący egzemplarz obcego dodamy na końcu metody `_init_()` w klasie `AlienInvasion`. Ostatecznie zostanie utworzona cała flota obcych, co będzie wymagało znacznie większej ilości kodu, więc dobrze jest zdefiniować nową metodę pomocniczą o nazwie `_create_fleet()`.

Kolejność metod w klasie nie ma znaczenia, o ile jest stosowana pewna konsekwencja podczas ich definiowania. Zdecydowałem się na umieszczenie metody `_create_fleet()` tuż przed `_update_screen()`, choć możesz wybrać inne miejsce w klasie `AlienInvasion`. Zacznię od zaimportowania klasy `Alien`.

Oto uaktualnione polecenia `import` w pliku `alien_invasion.py`.

*Plik alien\_invasion.py:*

---

```
--cięcie--  
from ship import Bullet  
from alien import Alien
```

---

Spójrz na uaktualnioną metodę `_init_()`.

*Plik alien\_invasion.py:*

---

```
def __init__(self):  
    --cięcie--  
    self.ship = Ship(self)  
    self.bullets = pygame.sprite.Group()  
    self.aliens = pygame.sprite.Group()  
  
    self._create_fleet()
```

---

Utworzmy grupę przeznaczoną do przechowywania floty obcych oraz wywołamy metodę `_create_fleet()`, która zostanie zdefiniowana za chwilę.

Oto kod nowej metody `_create_fleet()`.

Plik alien\_invasion.py:

---

```
def _create_fleet(self):
    """Utworzenie pełnej floty obcych."""
    # Utworzenie obcego.
    alien = Alien(self)
    self.aliens.add(alien)
```

---

W tej metodzie następuje utworzenie jednego egzemplarza klasy `Alien`, a następnie dodanie go do grupy przechowującej całą flotę. Obcy zostanie umieszczony w lewym górnym rogu ekranu, co jest doskonałym położeniem dla pierwszego obcego.

Aby wyświetlić obcego na ekranie, w metodzie `_update_screen()` trzeba wywołać metodę `draw()` grupy.

Plik alien\_invasion.py:

---

```
def _update_screen(self):
    --cięcie--
    self.ship.blitme()
    self.aliens.draw(self.screen)

    pygame.display.flip()
```

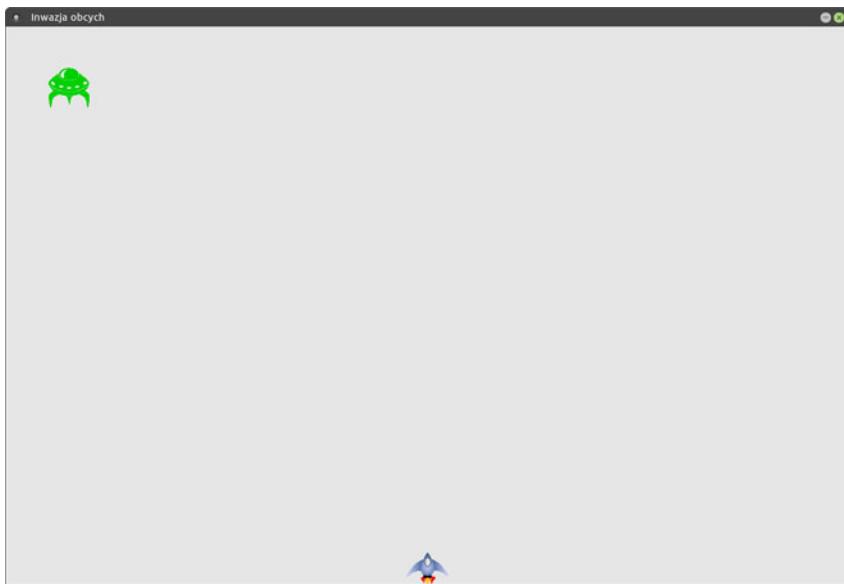
---

Podczas wywołania metody `draw()` w grupie Pygame wyświetla każdy element tej grupy w położeniu zdefiniowanym przez jego atrybut `rect`. Metoda `draw()` wymaga tylko jednego argumentu: powierzchni, na której mają zostać wyświetlone elementy grupy. Na rysunku 13.2 pokazałem pierwszego obcego na ekranie.

Skoro pierwszy obcy został prawidłowo wyświetlony na ekranie, możemy przystąpić do przygotowania kodu odpowiedzialnego za wyświetlenie całej floty obcych.

## Utworzenie floty obcych

Aby wyświetlić flotę, musimy ustalić, jaką liczbę obcych można zmieścić w jednym rzędzie. Można to zrobić na wiele sposobów. Przyjęte tutaj podejście polega na dodawaniu obcych u góry ekranu, aż w rzędzie zabraknie miejsca dla kolejnego obcego. Potem ten proces będzie powtarzany w następnych rzędach, aż na ekranie zabraknie miejsca na dodanie kolejnego rzędu obcych.



Rysunek 13.2. Na ekranie pojawił się pierwszy obcy

## Utworzenie rzędów obcych

Jesteś gotowy do wygenerowania rzędu pełnego obcych. W celu utworzenia pełnego rzędu obcych należy rozpocząć od umieszczenia pojedynczego obcego, aby mieć dostęp do jego szerokości. Obcy zostanie wyświetlony po lewej stronie ekranu, a następnie będziemy dodawać kolejnych, którzy zmieszcza się w rzędzie.

Plik alien\_invasion.py:

---

```
def _create_fleet(self):
    """Utworzenie pełnej floty obcych."""
    # Utworzenie obcego i dodawanie kolejnych obcych, którzy zmieszcza się w rzędzie.
    # Odległość między poszczególnymi obcymi jest równa szerokości obcego.
    alien = Alien(self)
    alien_width = alien.rect.width

    current_x = alien_width ❶
    while current_x < (self.settings.screen_width - 2 * alien_width): ❷
        new_alien = Alien(self) ❸
        new_alien.x = current_x ❹
        new_alien.rect.x = current_x
        self.aliens.add(new_alien)
        current_x += 2 * alien_width ❺
```

---

Z pierwszego utworzonego obcego pobieramy jego szerokość, a następnie definiujemy zmienną o nazwie `current_x` (patrz wiersz ❶). Odwołuje się ona do położenia poziomego następnego obcego, który ma być umieszczony na ekranie.

Początkowo ta wartość jest równa szerokości jednego obcego, co pozwala odsunąć pierwszego obcego floty od lewej krawędzi ekranu.

Następnie rozpoczynamy pętlę `while` (patrz wiersz ②), umożliwiającą dodawanie kolejnych obcych, którzy zmieszczą się w rzędzie. Aby ustalić, czy mamy miejsce na umieszczenie kolejnego obcego w rzędzie, wartość `current_x` trzeba porównać z wartością maksymalną. W trakcie pierwszej próby ta pętla może być zdefiniowania następująco:

---

```
while current_x < self.settings.screen_width:
```

---

Wydaje się, że to rozwiązańe działa, ale ostatni obcy w rzędzie będzie znajdował się już za prawą krawędzią ekranu. Dlatego konieczne jest dodanie niewielkiego marginesu po prawej stronie ekranu. Jeśli po prawej stronie będzie znajdowała się ilość miejsca równa dwóm szerokościom obcych, zostanie wykonana iteracja pętli, w trakcie której w rzędzie pojawi się kolejny obcy.

Gdy w rzędzie jest wystarczająca ilość miejsca do kontynuowania pętli, zostaną wykonane dwa zadania: utworzenie obcego we właściwym położeniu i zdefiniowanie poziomego położenia dla następnego obcego w rzędzie. Po utworzeniu nowego obcego jest on przypisywany zmiennej `new_alien` (patrz wiersz ③). Następnie definiujemy dokładne położenie poziome jako wartość `current_x` (patrz wiersz ④). Atrybut `x` obcego jest używany do określenia położenia jego prostokąta. Na końcu dodajemy nowego obcego do grupy `self.aliens`.

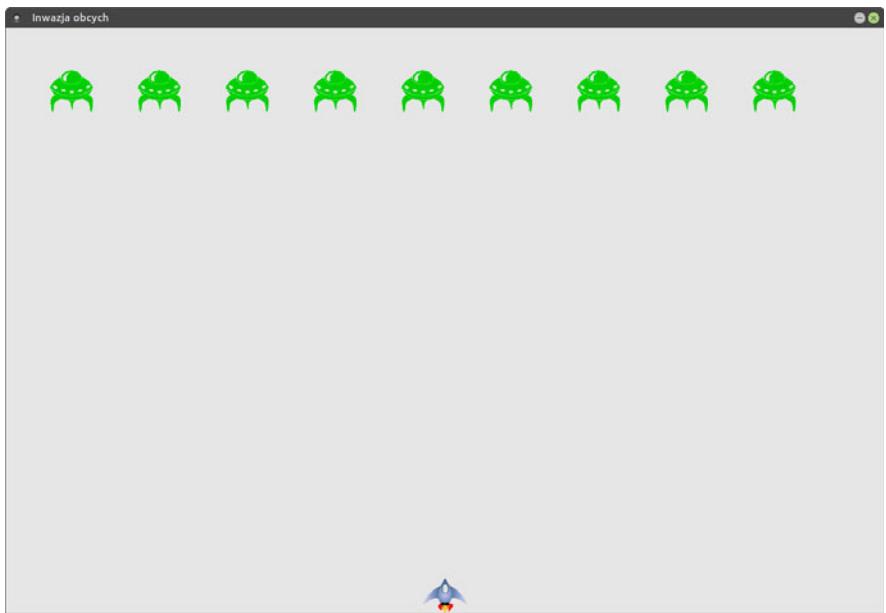
Następnie inkrementujemy wartość `current_x` (patrz wiersz ⑤). Do położenia poziomego dodajemy dwie szerokości obcych w celu przejścia poza nowo dodanego obcego i pozostawienia za nim pewnej ilości wolnego miejsca. Python ponownie sprawdzi warunek pętli `while` i ustali, czy jest miejsce pozwalające na utworzenie kolejnego obcego. Jeżeli miejsca nie ma, działanie pętli zostanie zakończone, a jej wynikiem będzie rząd pełny obcych.

Po uruchomieniu budowanej gry powinieneś zobaczyć na ekranie pierwszy rząd obcych, tak jak pokazalem na rysunku 13.3.

**UWAGA** *Nie zawsze będzie oczywiste, w jaki dokładnie sposób należy zdefiniować pętlę taką jak przedstawiona w tym podrozdziale. Jedną z zalet programowania jest to, że początkowe pomysły dotyczące podejścia do rozwiązania problemu nie muszą być poprawne. Jak najbardziej można utworzyć pętle, w których obcy zostaną umieszczeni zbyt daleko po prawej stronie, a następnie modyfikować tą pętlę aż do otrzymania oczekiwanej wyniku.*

## Refaktoryzacja metody `_create_fleet()`

Gdybyśmy zakończyli już tworzenie floty, prawdopodobnie pozostawilibyśmy metodę `_create_fleet()` w jej obecnej postaci. Ponieważ mamy jednak jeszcze trochę pracy do wykonania, przeprowadzimy teraz refaktoryzację tej metody. Dodamy nową metodę pomocniczą `_create_alien()`, która będzie wywoływana z poziomu `_create_fleet()`.



Rysunek 13.3. Pierwszy rzęd obcych

Plik alien\_invasion.py:

---

```
def _create_fleet(self):
    --cięcie--
    while current_x < (self.settings.screen_width - 2 * alien_width):
        self._create_alien(current_x)
        current_x += 2 * alien_width

def _create_alien(self, x_position): ❶
    """Utworzenie obcego i umieszczenie go w rzędzie."""
    new_alien = Alien(self)
    new_alien.x = x_position
    new_alien.rect.x = x_position
    self.aliens.add(new_alien)
```

---

Metoda `_create_alien()` wymaga tylko jednego parametru poza `self`: wartości współrzędnej  $X$  określającej położenie obcego (patrz wiersz ❶). Wykorzystany został ten sam kod, który wcześniej znajdował się w metodzie `_create_fleet()`, z tą jednak różnicą, że użyjemy parametru o nazwie `x_position` zamiast `current_x`. Dzięki przeprowadzonej tutaj refaktoryzacji dodawanie nowych rzędów i utworzenie całej floty obcych będzie łatwiejsze.

## Dodawanie rzędów

Aby ukończyć pracę nad flotą, konieczne jest dodawanie kolejnych rzędów, aż na ekranie zabraknie miejsca na następny rząd. W tym celu użyjemy zagnieżdzonej pętli — dotyczącej pętla `while` opakujemy nową. Zadaniem pętli wewnętrznej będzie poziome umieszczanie obcych w rzędzie, dzięki skoncentrowaniu się na wartości współrzędnych  $X$  obcych. Natomiast pętla zewnętrzna zajmuje się pionowym umieszczaniem obcych, koncentrując się na wartości współrzędnych  $Y$  obcych. Dodawanie rzędów zakończy się, gdy prawie dotrzymy do dolnej krawędzi ekranu. Pozostawione będzie wolne miejsce dla statku gracza oraz pozwalające na rozpoczęcie strzelania pociskami do obcych.

Oto jak przedstawiają się zagnieżdżone pętle `while` w metodzie `_create_fleet()`:

Plik alien\_invasion.py:

```
def _create_fleet(self):
    """Utworzenie pełnej floty obcych."""
    # Utworzenie obcego i dodawanie kolejnych obcych, którzy zmiesią się w rzędzie.
    # Odległość między poszczególnymi obcymi jest równa szerokości obcego.
    alien = Alien(self)
    alien_width, alien_height = alien.rect.size ❶

    current_x, current_y = alien_width, alien_height ❷
    while current_y < (self.settings.screen_height - 3 * alien_height): ❸
        while current_x < (self.settings.screen_width - 2 * alien_width):
            self._create_alien(current_x, current_y) ❹
            current_x += 2 * alien_width

    # Ukończenie rzędu, wyzerowanie wartości x oraz inkrementacja wartości y. ❺
    current_x = alien_width
    current_y += 2 * alien_height
```

W celu umieszczania rzędów na ekranie potrzebna jest wysokość obcego, więc w wierszu ❶ używamy atrybutu `size`, który reprezentuje szerokość i wysokość obiektu `rect`. Ten atrybut jest krotką zawierającą szerokość i wysokość.

Następnie definiujemy początkowe wartości współrzędnych  $X$  i  $Y$  do umieszczania pierwszego obcego z floty (patrz wiersz ❷). Obcy zostaje odsunięty od lewej krawędzi ekranu o odległość odpowiadającą szerokości obcego, a od górnej krawędzi ekranu o odległość równą wysokości obcego. Kolejnym krokiem jest zdefiniowanie pętli `while`, która kontrole liczbę rzędów umieszczanych na ekranie (patrz wiersz ❸). Jeżeli wartość  $Y$  dla następnego rzędu jest mniejsza niż wysokość ekranu pomniejszona o trzy wysokości obcego, kontynuujemy dodawanie rzędów. (Jeśli nie zostanie pozostawiona odpowiednia ilość miejsca, można to później zmienić).

Wywołujmy metodę `_create_alien()` i przekazujemy jej wartość  $Y$  oraz położenie  $X$  (patrz wiersz ❹). Metodę `_create_alien()` zmodyfikujemy za chwilę.

Zwróci uwagę na wcięcie dwóch ostatnich wierszy kodu ❺. Znajdują się one w zewnętrznej pętli `while`, ale poza wewnętrzną pętlą `while`. Ten blok kodu zostanie

wykonany po zakończeniu działania pętli wewnętrznej. Te dwa polecenia będą wykonywane jednokrotnie dla każdego utworzonego rzędu obcych. Po dodaniu każdego rzędu następuje wyzerowanie wartości `current_x`, aby pierwszy obcy w następnym rzędzie został umieszczony w tym samym położeniu co w poprzednich rzędach. Następnie do bieżącej wartości `current_y` została dodana wartość odpowiadająca dwóm wysokościom obcego, aby następny rząd obcych znajdował się nieco niżej na ekranie. W tym miejscu wspomniane wcięcia naprawdę mają duże znaczenie. Jeżeli po uruchomieniu pliku `alien_invasion.py` na końcu tego podrozdziału nie otrzymasz poprawnej floty, sprawdź wcięcia wszystkich wierszy w tych zagnieżdżonych pętlach.

Konieczne jest zmodyfikowanie metody `_create_alien()`, aby poprawnie zdefiniować pionowe położenie obcego.

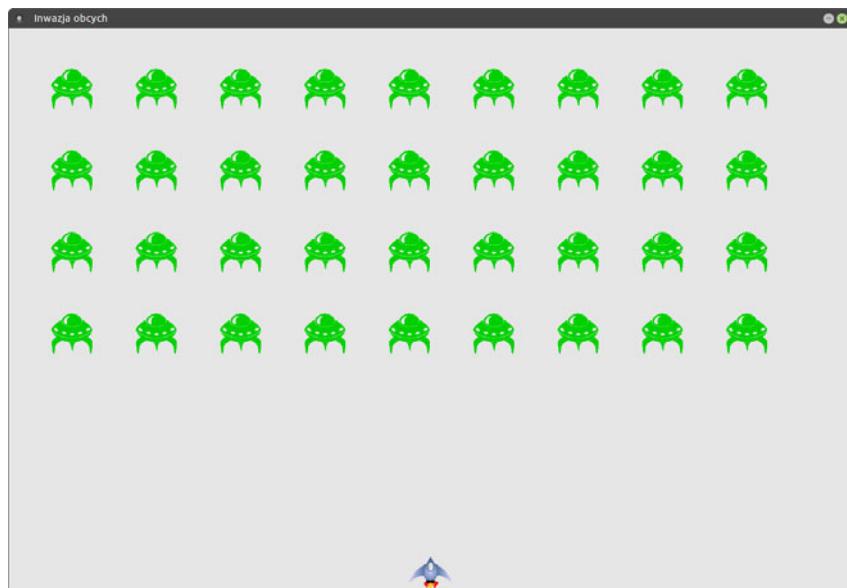
---

```
def _create_alien(self, x_position, y_position):
    """Utworzenie obcego i umieszczenie go w rzędzie."""
    new_alien = Alien(self)
    new_alien.x = x_position
    new_alien.rect.x = x_position
    new_alien.rect.y = y_position
    self.aliens.add(new_alien)
```

---

Modyfikujemy definicję metody, aby akceptowała wartość Y dla nowego obcego. Ponadto ustalamy położenie pionowe prostokąta obcego.

Jeżeli teraz uruchomisz grę, powinieneś zobaczyć pełną flotę obcych, tak jak pokazalem na rysunku 13.4.



Rysunek 13.4. Pełna flota obcych w budowanej grze

W kolejnym podrozdziale wprowadzimy tę flotę w ruch!

## Poruszanie flotą obcych

Teraz wprawimy flotę obcych w ruch. Najpierw będzie poruszała się w prawą stronę aż do dotarcia do krawędzi ekranu, a następnie przesunie się o pewną odległość w dół i zacznie poruszać się w przeciwnym kierunku. Ten ruch będzie kontynuowany aż do chwili zestrzelenia wszystkich obcych przez gracza, zderzenia się obcego ze statkiem kosmicznym kierowanym przez gracza lub dotarcia choć jednego obcego do dolnej krawędzi ekranu. Pracę rozpoczynamy od zaprogramowania ruchu floty w prawą stronę.

### ZRÓB TO SAM

**13.1. Gwiazdy.** Wyszukaj obraz przedstawiający gwiazdę. Wyświetl na ekranie całą siatkę gwiazd.

**13.2. Jeszcze lepsze gwiazdy.** Możesz przygotować jeszcze bardziej rzeczywiste wzorce gwiazd przez zastosowanie pewnego czynnika losowości podczas umieszczania gwiazd na ekranie. Poniżej przypominam, w jaki sposób można wygenerować losową liczbę:

```
from random import randint  
random_number = randint(-10,10)
```

Wartością zwrotną powyższego fragmentu kodu jest liczba całkowita z zakresu od -10 do 10. Wykorzystując kod z ćwiczenia 13.1, zmień położenie każdej gwiazdy o losowo wygenerowaną liczbę.

## Przesunięcie obcych w prawo

Aby obcy mogli się poruszać, wykorzystujemy metodę `update()` w pliku `alien.py`. Metoda ta będzie wywoływana dla każdego obcego w grupie. Przede wszystkim musimy dodać ustawienie pozwalające na określenie szybkości poruszania się obcego.

Plik `settings.py`:

```
def __init__(self):  
    --cięcie--  
    # Ustawienia dotyczące obcego.  
    self.alien_speed = 1.0
```

Następnie nowo dodaną opcję wykorzystujemy do implementacji metody `update()`.

*Plik alien.py:*

---

```
def __init__(self, ai_game):
    """Inicjalizacja obcego i umieszczenie go w położeniu początkowym."""
    super().__init__()
    self.screen = ai_game.screen
    self.settings = ai_game.settings
    --cięcie--

def update(self):
    """Przesunięcie obcego w prawo."""
    self.x += self.settings.alien_speed ①
    self.rect.x = self.x ②
```

---

Parametr ustawień jest tworzony w metodzie `__init__()` i dzięki temu dostęp do szybkości poruszania się obcego mamy w metodzie `update()`. W trakcie każdego uaktualnienia położenia obcego przesuwamy go w prawo o odległość wskazywaną przez `alien_speed`. Dokładne położenie obcego monitorujemy za pomocą atrybutu `self.x`, który może przechowywać wartości zmiennoprzecinkowe (patrz wiersz **①**). Następnie wartości `self.x` używamy do uaktualnienia położenia prostokąta (`rect`) obcego (patrz wiersz **②**).

Na razie w głównej pętli `while` mamy wywołania uaktualniające położenie statku oraz pocisków. Konieczne jest również uaktualnienie położenia każdego obcego.

*Plik alien\_invasion.py:*

---

```
while True:
    self._check_events()
    self.ship.update()
    self._update_bullets()
    self._update.aliens()
    self._update_screen()
    self.clock.tick(60)
```

---

Przygotujemy kod odpowiedzialny za zarządzanie ruchem floty i umieścimy go w nowej metodzie o nazwie `_update_aliens()`. Położenie obcych jest uaktualniane po uaktualnieniu położenia pocisków, ponieważ wkrótce będziemy sprawdzać, czy któryś pocisk trafił w jakiegoś obcego.

Miejsce umieszczenia tej metody w module nie ma znaczenia. Jednak dla zachowania pewnej organizacji kodu zdecydowałem się na jej umieszczenie tuż po `_update_bullets()` i tym samym zachowanie kolejności wywołań metod w pętli `while`. Spójrz na pierwszą wersję metody `_update_aliens()`.

*Plik alien\_invasion.py:*

---

```
def _update_aliens(self):
    """Uaktualnienie położenia wszystkich obcych we flocie."""
    self.aliens.update()
```

---

Metodę `update()` wywołujemy dla grupy `aliens`, co automatycznie spowoduje wywołanie metody `update()` dla każdego egzemplarza obcego. Jeżeli teraz uruchomisz budowaną grę, zobaczysz, że flota obcych porusza się w prawą stronę i znika za krawędzią ekranu.

## Zdefiniowanie ustawień dla kierunku poruszania się floty

Przystępujemy teraz do zdefiniowania ustawień, które spowodują poruszanie się floty w dół ekranu oraz zwrot w lewą stronę po dotarciu do prawej krawędzi ekranu. Oto kod implementujący takie zachowanie floty.

Plik `settings.py`:

---

```
# Ustawienia dotyczące obcego.
self.alien_speed = 1.0
self.fleet_drop_speed = 10
# Wartość fleet_direction wynosząca 1 oznacza prawo, natomiast -1 oznacza lewo.
self.fleet_direction = 1
```

---

Ustawienie `fleet_drop_speed` określa szybkość, z jaką flota będzie poruszać się w dół ekranu po dotarciu do jego prawej lub lewej krawędzi. Dobrze jest oddzielić tę szybkość od szybkości poruszania się obcych w poziomie, ponieważ w ten sposób będziemy mogli je niezależnie regulować.

Aby zaimplementować ustawienie `fleet_direction`, moglibyśmy wykorzystać wartość tekstową taką jak `left` lub `right`, ale to oznaczałoby konieczność przygotowania poleceń `if-elif` sprawdzających kierunek poruszania się floty. Ponieważ musimy zapewnić obsługę jedynie dwóch kierunków, użyjemy wartości `1` i `-1` do zmiany kierunku poruszania się floty. (Użycie liczb ma sens z jeszcze jednego powodu — ruch w prawą stronę oznacza dodanie wartości do współrzędnej X obcego, natomiast ruch w lewą stronę oznacza odjęcie wartości od współrzędnej X obcego).

## Sprawdzenie, czy obcy dotarł do krawędzi ekranu

Potrzebna jest nam metoda, dzięki której sprawdzimy, czy obcy dotarł do krawędzi ekranu. Zmodyfikujemy również metodę `update()`, aby pozwolić każdemu obcowi na poruszanie się w odpowiednim kierunku. Przedstawiony tutaj kod jest częścią klasy `Alien`.

Plik `alien.py`:

---

```
def check_edges(self):
    """Zwraca wartość True, jeśli obcy znajduje się przy krawędzi ekranu."""
    screen_rect = self.screen.get_rect()
    return (self.rect.right >= screen_rect.right) or (self.rect.left <= 0) ❶

def update(self):
```

```
    """Przesunięcie obcego w prawo lub w lewo."""
    self.x += self.settings.alien_speed * ②
        self.settings.fleet_direction
    self.rect.x = self.x
```

---

Teraz nową metodę `check_edges()` możemy wywoływać dla każdego obcego, aby sprawdzić, czy znajduje się on przy lewej lub prawej krawędzi ekranu. Obcy będzie przy prawej krawędzi ekranu, gdy atrybut `right` jego prostokąta (`rect`) będzie miał wartość większą niż lub równą wartości atrybutu `right` prostokąta ekranu. Obcy znajduje się przy lewej krawędzi ekranu, gdy wartość `left` jest mniejsza niż lub równa zero (patrz wiersz ①). Zamiast umieszczać to sprawdzenie w bloku `if`, warunek zostaje zdefiniowany bezpośrednio w poleceniu `return`. Ta metoda zwraca wartość `True`, jeśli obcy znajduje się przy lewej bądź prawej krawędzi ekranu, a w innych przypadkach jej wartością zwrotną jest `False`.

Aby flota mogła poruszać się w prawą lub w lewą stronę, modyfikujemy metodę `update()` przez pomnożenie szybkości obcego i wartości przechowywanej we `fleet_direction` (patrz wiersz ②). Jeżeli wartością `fleet_direction` jest 1, wartość `alien_speed` zostanie dodana do bieżącego położenia obcego, który w ten sposób przesunie się w prawą stronę. Natomiast wartość `fleet_direction` wynoszącą -1 oznacza odjęcie wartości `alien_speed` od bieżącego położenia obcego, który w ten sposób przesunie się w lewą stronę.

## Przesunięcie floty w dół i zmiana kierunku

Kiedy obcy dotrze do krawędzi ekranu, cała flota musi przesunąć się w dół i zmienić kierunek, w którym się porusza. Konieczne jest więc wprowadzenie znaczących zmian w klasie `AlienInvasion`, ponieważ będziemy sprawdzać, czy którykolwiek obcy dotarł do prawej lub lewej krawędzi ekranu. Aby zaimplementować takie rozwiązanie, przygotujemy jeszcze dwie metody: `_check_fleet_edges()` i `_change_fleet_direction()`, a także zmodyfikujemy metodę `_update.aliens()`. Nowe metody umieściłem po `_create_alien()`, choć tak naprawdę ich położenie w klasie nie ma znaczenia.

Plik `alien_invasion.py`:

```
def _check_fleet_edges(self):
    """Odpowiednia reakcja, gdy obcy dotrze do krawędzi ekranu."""
    for alien in self.aliens.sprites(): ①
        if alien.check_edges():
            self._change_fleet_direction() ②
            break

def _change_fleet_direction():
    """Przesunięcie całej floty w dół i zmiana kierunku, w którym się ona porusza."""
    for alien in self.aliens.sprites():
        alien.rect.y += self.settings.fleet_drop_speed ③
        self.settings.fleet_direction *= -1
```

---

W metodzie `_check_fleet_edges()` przeprowadzamy iterację przez flotę i wywołujemy metodę `check_edges()` dla każdego obcego (patrz wiersz ❶). Jeżeli wartością zwrotną `check_edges()` jest `True`, wówczas wiemy, że obcy znajduje się przy krawędzi ekranu i cała flota powinna zmienić kierunek. Wywołujemy więc `_change_fleet_direction()` i kończymy działanie pętli (patrz wiersz ❷). W metodzie `_change_fleet_direction()` przeprowadzamy iterację przez wszystkich obcych i każdego z nich przesuwamy w dół za pomocą ustawienia `fleet_drop_speed` (patrz wiersz ❸). Następnym krokiem jest zmiana wartości `fleet_direction`, co odbywa się przez pomnożenie bieżącej wartości tej zmiennej przez `-1`. Wiersz zawierający polecenie zmieniające kierunek poruszania się floty nie jest częścią pętli `for`. Chcemy zmienić położenie pionowe każdego obcego, natomiast kierunek poruszania się całej floty chcemy zmienić tylko raz.

Spójrz na zmodyfikowaną wersję metody `_update.aliens()`.

Plik alien\_invasion.py:

```
def _update.aliens(self):
    """
    Sprawdzenie, czy flota obcych znajduje się przy krawędzi,
    a następnie uaktualnienie położenia wszystkich obcych we flocie.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

Zmodyfikowaliśmy metodę przez wywołanie `_check_fleet_edges()` przed uaktualnieniem położenia poszczególnych obcych.

Jeżeli teraz uruchomisz grę, flota powinna poruszać się od jednej krawędzi ekranu do drugiej i przesuwać się w dół po każdym dotarciu do krawędzi. Możemy więc rozpocząć zestrzeliwanie obcych i wypatrywać obcego, który zetknie się z kierowanym przez gracza statkiem kosmicznym lub dotrze do dolnej krawędzi ekranu.

## ZRÓB TO SAM

**13.3. Krople deszczu.** Wyszukaj obraz przedstawiający kropę deszczu, a następnie za jego pomocą utwórz siatkę składającą się z wielu kropli. Te kropki powinny spadać w kierunku dolnej krawędzi ekranu i tam znikać.

**13.4. Opad deszczu.** Pracę rozpoczęj od kodu utworzonego w poprzednim ćwiczeniu. Kiedy rząd kropli deszczu zniknie za dolną krawędzią ekranu, nowy wiersz kropli powinien pojawić się na górze ekranu i zacząć opadać w dół.

# Zestrzeliwanie obcych

Zbudowaliśmy statek kosmiczny i flotę obcych, ale gdy wystrzelony pocisk dotrze do obcego, po prostu przechodzi przez niego, ponieważ nie zaimplementowaliśmy jeszcze wykrywania kolizji. W programowaniu gier tak zwana *kolizja* występuje, gdy elementy gry nakładają się na siebie. Dlatego też aby wystrzelony pocisk spowodował zniszczenie obcego, musimy wykorzystać metodę `sprite.groupcollide()` w celu wyszukania kolizji między elementami składowymi dwóch grup.

## Wykrywanie kolizji z pociskiem

Kiedy pocisk trafi w obcego, chcemy się o tym natychmiast dowiedzieć i sprawić, że obcy jak najszybciej zniknie z ekranu. W tym celu sprawdzenie pod względem kolizji będzie przeprowadzane zaraz po aktualnieniu położenia pocisku.

Metoda `sprite.groupcollide()` porównuje prostokąt każdego pocisku z prostokątem każdego obcego, a następnie zwraca słownik zawierający kolidujące ze sobą pociski i pojazdy obcych. Kluczem w słowniku jest pocisk, natomiast wartością trafiony przez niego obcy. (Ten słownik wykorzystamy także podczas tworzenia w rozdziale 14. systemu punktacji w grze).

Przedstawiony poniżej kod służy do sprawdzenia kolizji w metodzie `_update_bullets()`.

Plik alien\_invasion.py:

---

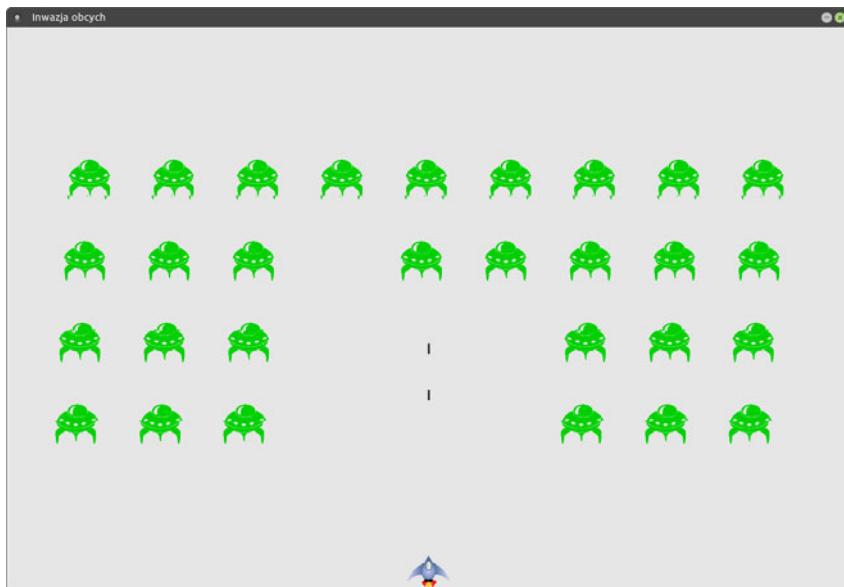
```
def _update_bullets(self):
    """Uaktualnienie położenia pocisków i usunięcie niewidocznych na ekranie."""
    --cięcie--

    # Sprawdzenie, czy którykolwiek pocisk trafil obcego.
    # Jeżeli tak, usuwamy zarówno pocisk, jak i obcego.
    collisions = pygame.sprite.groupcollide(
        self.bullets, self.aliens, True, True)
```

---

Nowy wiersz kodu dodany do powyższej metody powoduje przeprowadzenie iteracji przez wszystkie pociski znajdujące się w grupie `self.bullets` oraz iteracji przez wszystkich obcych umieszczonych w grupie `self.aliens`. Jeżeli prostokąty któregokolwiek pocisku i któregokolwiek obcego się nakładają, metoda `groupcollide()` dodaje parę klucz-wartość do zwracanego słownika. Dwa argumenty `True` wskazują bibliotece Pygame, czy należy usunąć pocisk i obcego, jeśli ze sobą kolidują. (W celu utworzenia superpocisku, który porusza się do góry ekranu, niszcząc wszystkich obcych na swojej ścieżce, pierwszemu argumentowi booleanskiemu możesz przypisać wartość `False`, a drugiemu pozostawić `True`. W ten sposób trafiony obcy zniknie, natomiast pocisk pozostanie aktywny aż do chwili wyjścia poza ekran).

Kiedy uruchomisz budowaną grę, trafiony obcy powinien zniknąć z ekranu. Na rysunku 13.5 pokazalem flotę, w której część obcych została już zestrzelona.



Rysunek 13.5. Wreszcie możemy zestrzeliwać obcych!

## Utworzenie większych pocisków w celach testowych

Wiele funkcji gry można przetestować, po prostu uruchamiając grę i je wypróbowując. Jednak niektóre z funkcjonalności będą niezwykle uciążliwe do przetestowania w normalnej wersji gry. Na przykład dużo pracy wymaga wielokrotne zestrzelenie wszystkich obcych na ekranie, aby sprawdzić, czy kod na pewno prawidłowo reaguje na pustą flotę.

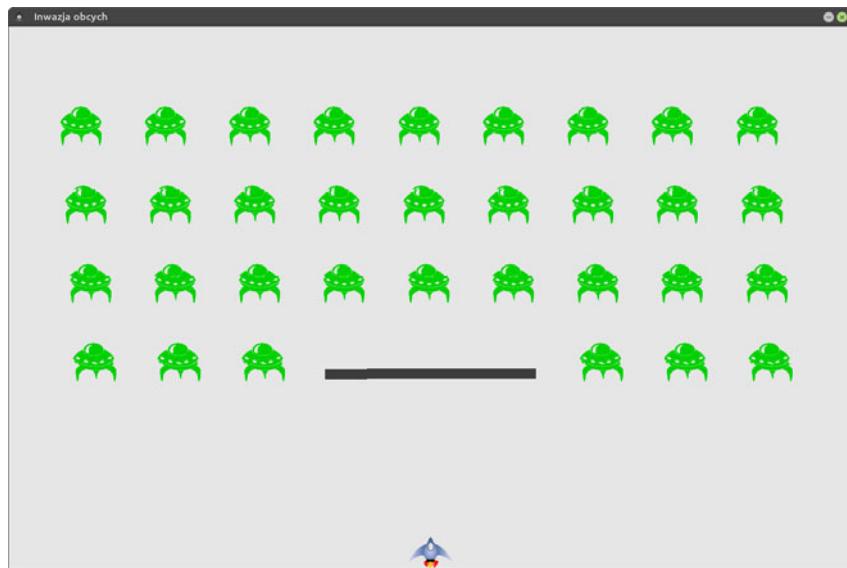
W celu przetestowania określonych funkcji można zmienić pewne ustawienia w grze i skoncentrować się na wybranych obszarach. Na przykład jedną z możliwości jest zwężenie ekranu i tym samym zmniejszenie liczby obcych do zestrzelania, lub też zwiększenie szybkości pocisków i dostarczenie większej ich liczby.

Moją ulubioną zmianą wprowadzaną podczas testowania gry *Inwazja obcych* jest superszeroki pocisk, który pozostaje aktywny nawet po zestrzeleniu obcego (patrz rysunek 13.6). Spróbuj przypisać atrybutowi `bullet_width` wartość 300 lub nawet 3000, a zobaczysz, jak szybko możesz zestrzelić całą flotę!

Tego rodzaju zmiana pomoże znacznie efektywniej przetestować grę i prawdopodobnie podsumie pewne pomysły dotyczące oferowania graczy bonusów. Pamiętaj o przywróceniu początkowych ustawień po zakończeniu testowania funkcjonalności gry.

## Ponowne utworzenie floty

Jedną z najważniejszych cech gry *Inwazja obcych* jest to, że liczba obcych jest nieograniczona. Kiedy zniszczysz jedną flotę, w jej miejsce pojawi się następna.



Rysunek 13.6. Niezwykle potężny pocisk bardzo ułatwia testowanie niektórych aspektów gry

Aby nowa flota obcych pojawiła się w miejscu floty zniszczonej przez gracza, w pierwszej kolejności trzeba sprawdzić, czy grupa aliens jest pusta. Jeżeli tak, wywołujemy metodę `_create_fleet()`. Operację sprawdzenia przeprowadzamy w `_update_bullets()`, ponieważ to właśnie tutaj znajduje się kod odpowiedzialny za niszczenie poszczególnych obcych.

Plik `alien_invasion.py`:

---

```
def _update_bullets(self):
    --cięcie--
    if not self.aliens: ❶
        # Pozbycie się istniejących pocisków i utworzenie nowej floty.
        self.bullets.empty() ❷
        self._create_fleet()
```

---

W wierszu ❶ sprawdzamy, czy grupa `aliens` jest pusta. Ponieważ wartością pustej grupy jest `False`, w omawianym przykładzie sprawdzamy, czy grupa jest pusta. Jeżeli tak, pozbywamy się istniejących pocisków za pomocą metody `empty()`, która usuwa wszystkie pozostałe sprite'y w grupie (patrz wiersz ❷). Ponadto wywołujemy metodę `_create_fleet()`, aby na ekranie ponownie wyświetlić flotę obcych.

Teraz nowa flota pojawi się na ekranie zaraz po tym, jak dotychczasowa flota zostanie całkowicie zniszczona.

## Zwiększenie szybkości pocisku

Jeżeli próbowałeś zestrzelić obcych na obecnym etapie gry, na pewno zauważysz, że szybkość poruszania się pocisków znaczco spadła. W Twoim komputerze mogą poruszać się nieco za wolno lub o wiele za szybko. Na tym etapie masz możliwość modyfikacji ustawień, aby rozgrywka była bardziej interesująca i przyjemna. Pamiętaj, że tempo gry będzie stopniowo się zwiększało, więc na początku nie powinno być zbyt szybkie.

Możemy zwiększyć szybkość pocisków, modyfikując wartość `bullet_speed` w pliku `settings.py`. Kiedy w moim komputerze wartość ta zostanie zwiększona do 2.5, pociski poruszają się po ekranie nieco szybciej.

*Plik settings.py:*

---

```
# Ustawienia dotyczące pocisku.  
self.bullet_speed = 2.5  
self.bullet_width = 3  
--cięcie--
```

---

Najlepsza wartość dla tego ustawienia zależy od szybkości działania Twojego systemu, więc będziesz musiał nieco poeksperymentować. Możesz zmodyfikować także pozostałe ustawienia.

## Refaktoryzacja metody `_update_bullets()`

Przeprowadzimy teraz refaktoryzację metody `_update_bullets()`, aby nie wykonywała tak wielu różnych zadań. Kod odpowiedzialny za obsługę kolizji pocisku z obcym przeniesiemy do oddzielnej metody.

*Plik alien\_invasion.py:*

---

```
def _update_bullets(self):  
    --cięcie--  
    # Usunięcie pocisków, które znajdują się poza ekranem.  
    for bullet in self.bullets.copy():  
        if bullet.rect.bottom <= 0:  
            self.bullets.remove(bullet)  
  
    self._check_bullet_alien_collisions()  
  
def _check_bullet_alien_collisions(self):  
    """Reakcja na kolizję między pociskiem i obcym."""  
    # Usunięcie wszystkich pocisków i obcych, między którymi doszło do kolizji.  
    collisions = pygame.sprite.groupcollide(  
        self.bullets, self.aliens, True, True)  
  
    if not in self.aliens:  
        # Pozycje się istniejących pocisków i utworzenie nowej floty.  
        self.bullets.empty()  
        self._create_fleet()
```

---

Utworzyliśmy nową metodę o nazwie `_check_bullet_alien_collision()`, aby móc wyszukiwać kolizje między pociskami i obcymi, a także by podjąć odpowiednie działania po zniszczeniu całej floty obcych. W ten sposób zachowujemy prostotę funkcji `_update_bullets()` i ułatwiamy sobie dalszą rozbudowę gry.

## ZRÓB TO SAM

**13.5. Strzelanie na boki 2.** Przeszedłeś długą drogę od ćwiczenia 12.6, w którym utworzyłeś grę pozwalającą graczyowi strzelać na boki. W tym ćwiczeniu spróbuj rozbudować tę grę do postaci, w jakiej obecnie znajduje się gra *Inwazja obcych*. Dodaj flotę obcych i zdefiniuj jej ruch w kierunku statku kosmicznego kierowanego przez gracza. Ewentualnie utwórz kod umieszczający obcego w losowo wybranym położeniu po prawej stronie, a następnie kierujący go w stronę gracza. Dodaj kod powodujący zniknięcie obcego, gdy zostanie trafiony.

# Zakończenie gry

Jakiej przyjemności dostarczy gra, w której gracz nie może przegrać? Jeżeli gracz nie zestrzeli wystarczająco szybko floty obcych, zetknięcie obcego ze statkiem kierowanym przez gracza powinno go zniszczyć. Ograniczymy liczbę statków pozostających do dyspozycji gracza. Ponadto statek zostanie też utracony, gdy którykolwiek obcemu uda się dotrzeć do dolnej krawędzi ekranu. Zakończenie gry nastąpi po wykorzystaniu przez gracza wszystkich statków, którymi dysponuje.

## Wykrywanie kolizji między obcym i statkiem

Rozpoczynamy od wykrywania kolizji między obcym i statkiem kierowanym przez gracza, co pozwoli podjąć odpowiednie działania, kiedy obcy uderzy w statek. Sprawdzenie pod kątem kolizji jest przeprowadzane w klasie `AlienInvasion` natychmiast po uaktualnieniu położenia każdego obcego.

Plik `alien_invasion.py`:

---

```
def _update_aliens(self):
    --cięcie--
    self.aliens.update()

    # Wykrywanie kolizji między obcym i statkiem.
    if pygame.sprite.spritecollideany(self.ship, self.aliens): ❶
        print("Statek został trafiony!!!") ❷
```

---

Metoda `spritecollideany()` pobiera dwa argumenty, czyli sprite'a oraz grupę. Metoda sprawdza, czy którykolwiek z elementów składowych grupy ma kolizję z podanym sprite'em, i zatrzymuje iterację po wykryciu tego rodzaju kolizji.

W omawianym przykładzie przeprowadzana jest iteracja przez grupę aliens, a jej zakończenie następuje po wykryciu pierwszego obcego, który zderzył się ze statkiem kosmicznym kierowanym przez gracza.

Jeżeli nie występują żadne kolizje, wartością zwrotną spritecollideany() jest None i blok if nie zostanie wykonany (patrz wiersz ①). Natomiast kiedy zostanie wykryta kolizja między obcym i statkiem, wartością zwrotną jest egzemplarz obcego, który zderzył się ze statkiem. Wówczas nastąpi wykonanie bloku if i wyświetlenie komunikatu „Statek został trafiony!”, jak możesz zobaczyć w wierszu ②. Kiedy obcy uderzy w statek kosmiczny, będziemy musieli wykonać kilka zadań: usunąć wszystkich pozostałych obcych i pociski, ponownie umieścić statek kosmiczny na środku przy dolnej krawędzi ekranu oraz utworzyć nową flotę. Zanim przystąpimy do utworzenia kodu odpowiedzialnego za wymienione zadania, musimy sprawdzić, czy wybrane rozwiążanie, jeśli chodzi o wykrywanie kolizji między obcym i statkiem, działa prawidłowo. Przygotowanie wywołania print() to najprostszy sposób na upewnienie się, że mechanizm wykrywania kolizji spełnia swoją funkcję.

Kiedy uruchomimy budowaną grę, za każdym razem, gdy obcy będzie miał kolizję ze statkiem, w powłoce powinien zostać wyświetlony komunikat: „Statek został trafiony!”. Podeczas testowania tej funkcjonalności atrybutowi alien\_drop\_→speed przypisz większą wartość, na przykład 50 lub 100, aby obcy szybciej mógł dotrzeć do statku.

## Reakcja na kolizję między obcym i statkiem

Teraz musimy ustalić, co się będzie działało po wystąpieniu kolizji między obcym i statkiem kierowanym przez gracza. Zamiast usunąć egzemplarz ship i później utworzyć nowy, sprawdzamy liczbę kolizji między statkiem i obcymi, co odbywa się przez monitorowanie danych statystycznych dla gry. Monitorowanie tego rodzaju danych statystycznych jest użyteczne także podczas obsługi punktacji w grze.

Przystępujemy do utworzenia nowej klasy o nazwie GameStats przeznaczonej do monitorowania danych statystycznych. Kod tej klasy umieszczamy w pliku game\_stats.py.

Plik game\_stats.py:

---

```
class GameStats:
    """Monitorowanie danych statystycznych w grze „Inwazja obcych”. """

    def __init__(self, ai_game):
        """Inicjalizacja danych statystycznych."""
        self.settings = ai_game.settings
        self.reset_stats() ①

    def reset_stats(self):
        """
        Inicjalizacja danych statystycznych, które mogą zmieniać się w trakcie gry.
        """
        self.ships_left = self.settings.ship_limit
```

---

Tworzymy jeden egzemplarz GameStats, który będzie używany przez cały czas trwania gry. Musimy jednak mieć możliwość zerowania pewnych danych statystycznych za każdym razem, gdy gracz będzie rozpoczynał nową rozgrywkę. W tym celu większość danych statystycznych inicjalizujemy w metodzie `reset_stats()` zamiast bezpośrednio w `__init__()`. Metodę `reset_stats()` wywołujemy z `__init__()`, aby dane statystyczne były prawidłowo zdefiniowane po pierwszym utworzeniu egzemplarza GameStats (patrz wiersz ①). Oczywiście zachowujemy możliwość wywoływanego `reset_stats()` za każdym razem, gdy gracz rozpoczyna nową grę. Obecnie mamy tylko jeden rodzaj danych statystycznych — zmienną `ships_left`, której wartość zmienia się podczas gry.

Liczba statków, które są do dyspozycji gracza na początku gry, jest przechowywana w pliku `settings.py` jako `ship_limit`.

---

*Plik settings.py:*

---

```
# Ustawienia dotyczące statku.  
self.ship_speed = 1.5  
self.ship_limit = 3
```

---

Konieczne jest również wprowadzenie kilku zmian w pliku `alien_invasion.py`, aby utworzyć egzemplarz GameStats. Przede wszystkim na początku pliku należy dodać niezbędne polecenia import.

---

*Plik alien\_invasion.py:*

---

```
import sys  
from time import sleep  
  
import pygame  
  
from settings import Settings  
from game_stats import GameStats  
from ship import Ship  
--cięcie--
```

---

Importujemy funkcję `sleep()` z modułu `time` w bibliotece standardowej Pythona, co pozwoli zatrzymać grę na chwilę po trafieniu statku kosmicznego kierowanego przez gracza. Ponadto importujemy nową klasę `GameStats`.

W metodzie `__init__()` tworzymy egzemplarz klasy `GameStats`.

---

*Plik alien\_invasion.py:*

---

```
def __init__(self):  
    --cięcie--  
    self.screen = pygame.display.set_mode(  
        (self.settings.screen_width, self.settings.screen_height))  
    pygame.display.set_caption("Inwazja obcych")
```

```
# Utworzenie egzemplarza przechowującego dane statystyczne dotyczące gry.  
self.stats = GameStats(self)  
  
self.ship = Ship(self)  
--cięcie--
```

---

Egzemplarz tworzymy po przygotowaniu okna gry, ale jeszcze przed powstaniem pozostałych elementów gry, takich jak statek kosmiczny.

Kiedy obcy zderzy się ze statkiem kierowanym przez gracza, od liczby statków pozostałych gracowi odejmujemy wartość 1, usuwamy wszystkich istniejących obcych i pociski, tworzymy nową flotę obcych i ponownie umieszczamy statek na środku przy dolnej krawędzi ekranu. Wstrzymujemy także grę na chwilę, aby gracz mógł zauważać kolizję i przegrupowanie, zanim na ekranie pojawi się nowa flota obcych.

Większość wymienionego kodu umieścimy w nowej metodzie o nazwie `_ship_hit()`. Będzie ona wywoływana z metody `_update.aliens()` po trafieniu statku kosmicznego.

#### Plik alien\_invasion.py:

---

```
def _ship_hit(self):  
    """Reakcja na uderzenie obcego w statek."""  
    # Zmniejszenie wartości przechowywanej w ships_left.  
    self.stats.ships_left -= 1 1  
  
    # Usunięcie zawartości list bullets i aliens.  
    self.bullets.empty() 2  
    self.aliens.empty()  
  
    # Utworzenie nowej floty i wyśrodkowanie statku.  
    self._create_fleet() 3  
    self.ship.center_ship()  
  
    # Pauza.  
    sleep(0.5) 4
```

---

Nowa metoda `_ship_hit()` odpowiada za reakcję na kolizję między statkiem kierowanym przez gracza i obcym. Wewnątrz omawianej metody liczba statków pozostałych do dyspozycji gracza zostaje zmniejszona o 1 (patrz wiersz **1**), po czym następuje opróżnienie grup `aliens` i `bullets` (patrz wiersz **2**).

Następnym krokiem jest utworzenie nowej floty i umieszczenie statku kosmicznego na środku przy dolnej krawędzi ekranu (patrz wiersz **3**). Użytą tutaj metodę `center_ship()` za chwilę zaimplementujemy w klasie `Ship`. Po wprowadzeniu aktualnień w elementach gry, ale jeszcze przed wyświetleniem ich na ekranie, stosujemy pauzę, aby gracz mógł dostrzec, że kierowany przez niego statek zderzył się z obcym (patrz wiersz **4**). Ekran zostanie na chwilę zamrożony, a gracz będzie mógł zobaczyć, że obcy zderzył się z jego statkiem. Kiedy funkcja `sleep()` zakończy swoje działanie, nastąpi przejście do metody `_update_screen()`, która wyświetli na ekranie nową flotę obcych.

Uaktualniamy również definicję metody `_update_aliens()` i polecenie `print()` zastępujemy wywołaniem `_ship_hit()` po uderzeniu statku przez obcego.

#### Plik alien\_invasion.py:

---

```
def _update_aliens(self):
    --cięcie--
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()
```

---

Poniżej przedstawiłem nową metodę `center_ship()` — umieść ją na końcu pliku `ship.py`.

#### Plik ship.py:

---

```
def center_ship(self):
    """Umieszczenie statku na środku przy dolnej krawędzi ekranu."""
    self.rect.midbottom = self.screen_rect.midbottom
    self.x = float(self.rect.x)
```

---

Umieszczenie statku na środku odbywa się w dokładnie taki sam sposób jak wcześniej w metodzie `__init__()`. Po wyśrodkowaniu statku zerowana jest wartość atrybutu `self.x`, co pozwala śledzić dokładne położenie statku.

**UWAGA** Zwrć uwagę, że zawsze tworzymy tylko jeden statek. W trakcie całej gry mamy tylko jeden egzemplarz statku i umieszczamy go na środku przy dolnej krawędzi ekranu za każdym razem, gdy dojdzie do jego zderzenia z obcym. Dane statystyczne przechowywane w `ships_left` wskazują, czy gracz ma do dyspozycji jakikolwiek statek.

Uruchom grę, zestrzel kilku obcych i pozwól jednemu obcemu zderzyć się ze statkiem. Gra powinna zostać wstrzymana na chwilę, następnie powinna pojawić się nowa flota obcych, a statek kierowany przez gracza powinien zostać umieszczony na środku przy dolnej krawędzi ekranu.

## Obcy, który dociera do dolnej krawędzi ekranu

Jeżeli obcy dotrze do dolnej krawędzi ekranu, reakcja będzie dokładnie taka sama jak w przypadku zderzenia obcego ze statkiem kosmicznym kierowanym przez gracza. Dodaj nową metodę odpowiedzialną za przeprowadzenie tego rodzaju sprawdzenia. Ta metoda będzie zdefiniowana w pliku `alien_invasion.py`.

#### Plik alien\_invasion.py:

---

```
def _check_aliens_bottom(self):
    """Sprawdzenie, czy którykolwiek obcy dotarł do dolnej krawędzi ekranu."""
    for alien in self.aliens.sprites():
        if alien.rect.bottom >= self.settings.screen_height: ❶
```

---

```
# Tak samo jak w przypadku zderzenia statku z obcym.  
self._ship_hit()  
break
```

---

Metoda `_check.aliens_bottom()` sprawdza, czy którykolwiek obcy dotarł do dolnej krawędzi ekranu. Obcy znajdzie się przy tej krawędzi, gdy jego wartość `rect.bottom` będzie większa niż lub równa wysokości ekranu (patrz wiersz ❶). W przypadku gdy obcy dotrze do dolnej krawędzi ekranu, wywołujemy metodę `_ship_hit()`. Skoro jeden obcy dotarł do dolnej krawędzi ekranu, nie ma potrzeby sprawdzać pozostałych. Działanie pętli kończymy po wywołaniu `_ship_hit()`.

Nowa metoda jest wywoływaną z poziomu `_update.aliens()`.

*Plik alien\_invasion.py:*

---

```
def _update.aliens(self):  
    --cięcie--  
    # Wykrywanie kolizji między obcym i statkiem.  
    if pygame.sprite.spritecollideany(self.ship, self.aliens):  
        self._ship_hit()  
  
    # Wyszukiwanie obcych docierających do dolnej krawędzi ekranu.  
    self._check.aliens_bottom()
```

---

Metodę `_check.aliens_bottom()` wywołujemy po uaktualnieniu położenia wszystkich obcych oraz po sprawdzeniu, czy obcy zderzył się ze statkiem. Teraz nowa flota obcych zostanie wyświetlona na ekranie za każdym razem, gdy statek kierowany przez gracza zostanie uderzony przez obcego lub gdy obcy dotrze do dolnej krawędzi ekranu.

## Koniec gry!

Teraz można odnieść wrażenie, że gra *Inwazja obcych* jest kompletna, choć nigdy się nie kończy. Wartość `ships_left` po prostu staje się ujemna. Dodamy więc opcję `game_active` i wykorzystamy ją do wskazania, że gracz nie ma już do dyspozycji żadnych statków. Tę opcję należy umieścić na końcu metody `__init__()` w klasie `AlienInvasion`.

*Plik alien\_invasion.py:*

---

```
def __init__(self):  
    --cięcie--  
    # Uruchomienie gry „Inwazja obcych” w stanie aktywnym.  
    self.game_active = True
```

---

Teraz dodajemy kod do metody `_ship_hit()` przypisujący opcji `game_active` wartość `False`, jeżeli gracz wykorzystał już wszystkie dostępne dla niego statki.

*Plik alien\_invasion.py:*

---

```
def _ship_hit(self):
    """Reakcja na uderzenie obcego w statek."""
    if self.stats.ships_left > 0:
        # Zmniejszenie wartości przechowywanej w ships_left.
        self.stats.ships_left -= 1
        --cięcie--
        # Pauza.
        sleep(0.5)
    else:
        self.game_active = False
```

---

Większa część metody `_ship_hit()` pozostała niezmieniona. Cały istniejący kod został przeniesiony do bloku `if`, w którym następuje sprawdzenie, czy gracz ma do dyspozycji przynajmniej jeden statek. Jeżeli tak, tworzymy nową flotę, robimy pauzę i przechodzimy dalej. Natomiast jeśli gracz wykorzystał już wszystkie statki, opcji `game_active` przypisujemy wartość `False`.

## **Ustalenie, które komponenty gry powinny być uruchomione**

W pliku `alien_invasion.py` należy wskazać komponenty, które zawsze powinny być uruchomione, oraz te, które mają działać jedynie wtedy, gdy gra jest aktywna.

*Plik alien\_invasion.py:*

---

```
def run_game(self):
    """Rozpoczęcie pętli głównej gry."""
    while True:
        self._check_events()

        if self.game_active:
            self.ship.update()
            self._update_bullets()
            self._update_aliens()

        self._update_screen()
        self.clock.tick(60)
```

---

W pętli głównej zawsze trzeba wywołać metodę `_check_events()`, nawet jeśli gra pozostaje nieaktywna. Wynika to z tego, że nadal chcemy wiedzieć, czy gracz nacisnął na przykład klawisz `Q`, aby zakończyć działanie programu, lub też kliknął przycisk zamkający okno Pygame. Konieczne jest również uaktualnienie ekranu, aby móc wprowadzać zmiany na ekranie podczas oczekiwania na ewentualną decyzję gracza o rozpoczęciu nowej gry. Pozostała część wywołań metody jest niezbędna tylko podczas aktywnej gry, ponieważ w przeciwnym razie nie ma potrzeby uaktualniania położenia poszczególnych elementów gry.

Teraz, kiedy będziesz grał w *Inwazję obcych*, gra zostanie zakończona po wykorzystaniu wszystkich dostępnych statków.

### ZRÓB TO SAM

**13.6. Koniec gry.** Wykorzystaj kod utworzony w ćwiczeniu 13.5 i monitoruj informacje o tym, ile razy statek został uderzony przez obcych oraz ilu obcych udało się trafić graczowi. Określ i zaimplementuj odpowiedni warunek powodujący zakończenie gry.

## Podsumowanie

W tym rozdziale na przykładzie floty obcych dowiedziałeś się, jak utworzyć dużą liczbę identycznych elementów gry. Zobaczyłeś, jak można wykorzystać zagnieżdżone pętle do przygotowania siatki elementów oraz jak poruszać dużym zbiorem elementów gry przez wywołanie metody `update()` dla każdego z nich. Nauczyłeś się kontrolować kierunek poruszania się obiektów na ekranie i reagować na zdarzenia, takie jak flota obcych docierająca do krawędzi ekranu. Ponadto dowiedziałeś się, jak wykrywać kolizje i jakie podejmować działania, gdy pocisk trafi obcego lub obcy zderzy się ze statkiem kosmicznym kierowanym przez gracza. Na końcu rozdziału zobaczyłeś, jak można monitorować dane statystyczne w grze oraz jak używać opcji `game_active` do ustalenia, czy gra powinna się zakończyć.

W ostatnim rozdziale dotyczącym budowania gry *Inwazja obcych* zajmiemy się dodaniem przycisku *Gra*, pozwalającego graczowi na rozpoczęcie pierwszej gry lub ponowne rozpoczęcie gry po zakończeniu poprzedniej. Ponadto wprowadzimy zmianę polegającą na przyśpieszeniu nieco tempa gry po zestrzeleniu całej floty obcych oraz zaimplementujemy system punktacji. Ostatecznym wynikiem będzie w pełni wartościowa gra!

# 14

## Punktacja



W TYM ROZDZIALE ZAKOŃCZYMY PRACE NAD GRĄ *INWAZJA OBCYCH*. DODAMY PRZYCISK *Gra* POZWALAJĄCY NA ROZPOCZĘCIE GRY LUB JEJ WZNOWIENIE PO ZAKOŃCZENIU POPRZEDNIEJ ROZGRYWKI. WPROWADZIMY także pewne modyfikacje w samej grze, aby jej tempo zwiększało się po przejęciu gracza na wyższy poziom. Zaimportujemy również system punktacji. Po zakończeniu lektury rozdziału będziesz miał wystarczającą wiedzę, aby zacząć samodzielnie tworzyć gry, które będą się charakteryzowały poziomem trudności wzrastającym wraz z postępem poczynionym przez gracza podczas rozgrywki. Ponadto będziesz potrafił wyświetlać punktację.

### Dodanie przycisku Gra

W tym podrozdziale dodamy przycisk *Gra*, który wyświetlimy przed rozpoczęciem gry oraz ponownie po jej zakończeniu. Dzięki temu gracz będzie mógł znowu zagrać.

Na obecnym etapie prac gra rozpoczyna się tuż po uruchomieniu programu *alien\_invasion.py*. Uruchomimy więc grę w stanie nieaktywnym, a następnie poinformujemy użytkownika, że rozpoczęcie gry nastąpi po kliknięciu przycisku *Gra*. W metodzie `__init__()` klasy *AlienInvasion* wprowadź poniższe zmiany.

*Plik alien\_invasion.py:*

---

```
def __init__(self):
    """Inicjalizacja gry i utworzenie jej zasobów."""
    pygame.init()
    --cięcie--

    # Uruchomienie gry w stanie nieaktywnym.
    self.game_active = False
```

---

Teraz gra powinna rozpoczęć się w stanie nieaktywnym. Jednak dopóki nie przygotujemy przycisku *Gra*, użytkownik nie będzie miał żadnej możliwości rozpoczęcia gry.

## Utworzenie klasy Button

Ponieważ Pygame nie zawiera wbudowanej metody przeznaczonej do tworzenia przycisków, musimy przygotować klasę `Button`, aby zbudować prostokąt wypełniony kolorem i zawierający etykietę. Kod tej klasy będzie można później zastosować do utworzenia dowolnego przycisku w grze. Poniżej przedstawiłem pierwszą część klasy `Button`; kod umieść w pliku `button.py`.

*Plik button.py:*

---

```
import pygame.font

class Button:
    """Klasa przeznaczona do tworzenia przycisków dla gry."""

    def __init__(self, ai_game, msg): ❶
        """Inicjalizacja atrybutów przycisku."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()

        # Zdefiniowanie wymiarów i właściwości przycisku.
        self.width, self.height = 200, 50 ❷
        self.button_color = (0, 135, 0)
        self.text_color = (255, 255, 255)
        self.font = pygame.font.SysFont(None, 48) ❸

        # Utworzenie prostokąta przycisku i wyśrodkowanie go.
        self.rect = pygame.Rect(0, 0, self.width, self.height) ❹
        self.rect.center = self.screen_rect.center

        # Komunikat wyświetlany przez przycisk trzeba przygotować tylko jednokrotnie.
        self._prep_msg(msg) ❺
```

---

Zaczynamy od zainportowania modułu `pygame.font`, który pozwala Pygame wygenerować na ekranie tekst. Metoda `__init__()` pobiera parametry `self`, obiekt `ai_game` i egzemplarz `msg` zawierający tekst wyświetlany na przycisku (patrz wiersz ❶). W wierszu ❷ definiujemy wymiary przycisku, następnie jego kolor ustawiamy na zielony, a kolor tekstu wyświetlonego na przycisku na biały.

W wierszu ③ przygotowujemy atrybut font do wygenerowania tekstu. Argument None nakazuje Pygame użyć czcionki domyślnej, 48 to wyrażona w punktach wielkość czcionki. Aby wyśrodkować przycisk na ekranie, w wierszu ④ tworzymy prostokąt (rect) dla przycisku, a jego atrybutowi center przypisujemy wartość odpowiadającą ekranowi.

Pygame pracuje z tekstem, generując tekst w postaci obrazu przeznaczonego do wyświetlenia. W wierszu ⑤ wywołujemy metodę `_prep_msg()` w celu obsługi tej operacji generowania.

Poniżej przedstawiłem kod metody `_prep_msg()`.

*Plik button.py:*

---

```
def _prep_msg(self, msg):
    """
    Umieszczenie komunikatu w wygenerowanym obrazie i wyśrodkowanie tekstu
    na przycisku.
    """
    self.msg_image = self.font.render(msg, True, self.text_color, self.button_color) ❶
    self.msg_image_rect = self.msg_image.get_rect() ❷
    self.msg_image_rect.center = self.rect.center
```

---

Metoda `_prep_msg()` wymaga przekazania parametru `self` oraz tekstu przeznaczonego do wygenerowania jako obraz (`msg`). Wywołanie `font.render()` zamienia tekst przechowywany w `msg` na obraz, który następnie zostaje umieszczony w `msg_image` (patrz wiersz ❶). Metoda `font.render()` pobiera wartość boolowską wskazującą na zastosowanie wygładzania krawędzi czcionki (tak zwany *antialiasing*). Pozostałe argumenty określają kolory tekstu i tła. W omawianym przykładzie decydujemy się na zastosowanie wygładzania krawędzi czcionki (wartość `True`), a kolor tła ustawiamy na taki sam jak kolor przycisku. (Jeżeli nie podasz koloru tła, Pygame spróbuje wygenerować czcionkę na przezroczystym tle).

W wierszu ❷ wyśrodkowujemy obraz tekstu na przycisku, tworząc w tym celu prostokąt na podstawie obrazu tekstu i przypisując jego atrybutowi `center` wartość odpowiadającą przyciskowi.

Na koniec tworzymy metodę `draw_button()`, którą możemy wywołać, aby wyświetlić przycisk na ekranie.

*Plik button.py:*

---

```
def draw_button(self):
    # Wyświetlenie pustego przycisku, a następnie komunikatu na nim.
    self.screen.fill(self.button_color, self.rect)
    self.screen.blit(self.msg_image, self.msg_image_rect)
```

---

Wywołujemy metodę `screen.fill()`, aby wyświetlić prostokąt przedstawiający przycisk. Następne polecenie w powyższym fragmencie kodu to wywołanie metody `screen.blit()`, która jest odpowiedzialna za wyświetlenie obrazu tekstu na ekranie. Do tej metody zostanie przekazany obraz oraz obiekt `rect` powiązany z tym obrazem. W ten sposób kończymy pracę nad klasą `Button`.

## Wyświetlenie przycisku na ekranie

Klasę Button wykorzystamy do utworzenia przycisku *Gra* w klasie AlienInvasion. Najpierw trzeba uaktualnić sekcję zawierającą polecenia import.

Plik alien\_invasion.py:

```
--cięcie--  
from game_stats import GameStats  
from button import Button
```

Ponieważ jest potrzebny tylko jeden przycisk tego rodzaju, zdefiniujemy go bezpośrednio w metodzie `__init__()` klasy AlienInvasion. Przedstawiony tutaj kod umieść na samym końcu tej metody.

Plik alien\_invasion.py:

```
def __init__(self):  
    --cięcie--  
    self.game_active = False  
  
    # Utworzenie przycisku Gra.  
    self.play_button = Button(self, "Gra")
```

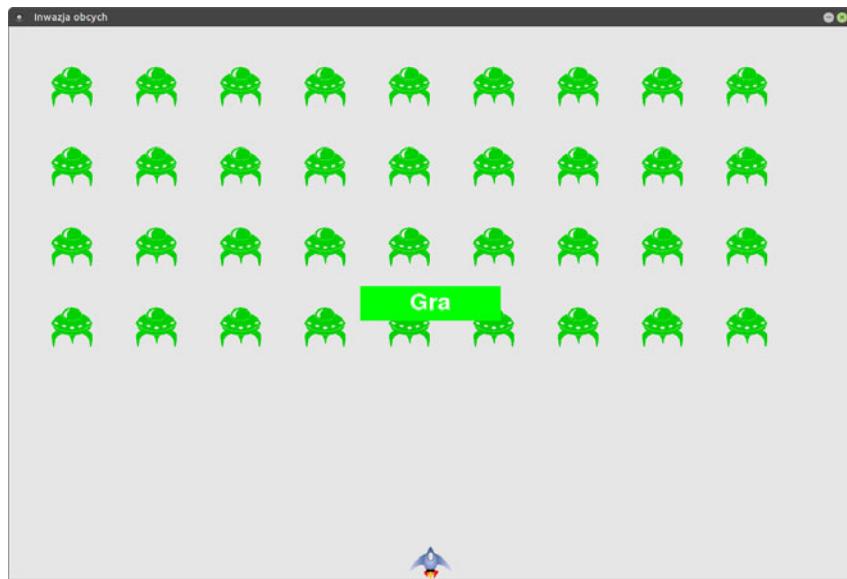
Działanie tego kodu polega na utworzeniu egzemplarza klasy Button wraz z etykietą *Gra*, choć ten przycisk jeszcze nie zostanie wyświetlony na ekranie. Metoda `draw_button()` zostanie wywołana z poziomu `_update_screen()`.

Plik alien\_invasion.py:

```
def _update_screen(self):  
    --cięcie--  
    self.aliens.draw(self.screen)  
  
    # Wyświetlenie przycisku tylko wtedy, gdy gra jest nieaktywna.  
    if not self.game_active:  
        self.play_button.draw_button()  
  
    pygame.display.flip()
```

Aby umieścić przycisk nad wszystkimi widocznymi elementami gry, wyświetlamy go po wygenerowaniu wszystkich pozostałych elementów gry, ale jeszcze przed przejściem do nowego ekranu. Wykorzystana została konstrukcja `if`, więc przycisk *Gra* będzie wyświetlany tylko wtedy, gdy gra jest nieaktywna.

Teraz po uruchomieniu budowanej gry powinieneś zobaczyć przycisk *Gra* wyświetlony na środku ekranu, tak jak pokazałem na rysunku 14.1.



Rysunek 14.1. Przycisk *Gra* będzie wyświetlany tylko wtedy, gdy gra będzie nieaktywna

## Uruchomienie gry

Aby po kliknięciu przycisku *Gra* następował uruchomienie nowej gry, przedstawiony poniżej kod umieść na końcu metody `_check_events()`. Zadanie tego bloku `elif` polega na monitorowaniu zdarzeń myszy nad przyciskiem.

Plik `alien_invasion.py`:

---

```
def _check_events(self):
    """Reakcja na zdarzenia generowane przez klawiaturę i mysz."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            --cięcie--
        elif event.type == pygame.MOUSEBUTTONDOWN: ❶
            mouse_pos = pygame.mouse.get_pos() ❷
            self._check_play_button(mouse_pos) ❸
```

---

Pygame wykrywa zdarzenie `MOUSEBUTTONDOWN` po kliknięciu przez gracza dowolnego punktu na ekranie (patrz wiersz ❶), ale w budowanej grze chcemy, aby reakcja została ograniczona jedynie do kliknąć w utworzony wcześniej przycisk *Gra*. Takie rozwiązanie wymaga użycia wywołania `pygame.mouse.get_pos()`, którego wartością zwrotną jest krotka zawierająca współrzędne X i Y kurSORA myszy po kliknięciu jej przycisku (patrz wiersz ❷). Te wartości w wierszu ❸ przekazujemy metodzie `_check_play_button()`.

Metodę `_check_play_button()` zdecydowałem się umieścić po metodzie `_check_events()`.

### Plik alien\_invasion.py:

---

```
def _check_play_button(self, mouse_pos):
    """Rozpoczęcie nowej gry po kliknięciu przycisku Gra przez użytkownika."""
    if self.play_button.rect.collidepoint(mouse_pos): ❶
        self.game_active = True
```

---

Metoda `collidepoint()` sprawdza, czy punkt kliknięcia myszy znajduje się na obszarze zdefiniowanym przez prostokąt przycisku *Gra* (patrz wiersz ❶). Jeżeli tak, atrybutowi `game_active` przypisujemy wartość `True` i gra się rozpoczyna!

Na tym etapie powinieneś mieć możliwość rozpoczęcia gry. Po jej zakończeniu wartość atrybutu `game_active` będzie wynosiła `False`, więc przycisk *Play* ponownie zostanie wyświetlony.

## Zerowanie gry

Przedstawiony powyżej kod sprawdza się doskonale w przypadku pierwszego kliknięcia przycisku *Gra* przez gracza, ale już nie po zakończeniu gry, ponieważ warunki, które spowodowały zakończenie gry, nie zostały wyzerowane.

Aby wyzerować grę po każdym kliknięciu przycisku *Gra*, konieczne jest wyzerowanie danych statystycznych gry, usunięcie wcześniej wyświetlanych obcych i pocisków, utworzenie nowej floty oraz wyśrodkowanie statku kosmicznego, jak pokazałem w poniższym fragmencie kodu.

### Plik alien\_invasion.py:

---

```
def _check_play_button(self, mouse_pos):
    """Rozpoczęcie nowej gry po kliknięciu przycisku Gra przez użytkownika."""
    if self.play_button.rect.collidepoint(mouse_pos):
        # Wyzerowanie danych statystycznych gry.
        self.stats.reset_stats() ❶
        self.game_active = True

        # Usunięcie zawartości list bullets i aliens.
        self.bullets.empty() ❷
        self.aliens.empty()

        # Utworzenie nowej floty i wyśrodkowanie statku.
        self._create_fleet() ❸
        self.ship.center_ship()
```

---

W wierszu ❶ zerujemy dane statystyczne gry, co oznacza, że gracz ponownie ma do dyspozycji trzy nowe statki. Atrybutowi `game_active` zostaje przypisana wartość `True` (więc gra rozpoczyna się po wykonaniu kodu w tej funkcji). Z grup `aliens` i `bullets` są usuwane wszystkie elementy (patrz wiersz ❷), tworzymy nową flotę oraz umieszczamy statek na środku przy dolnej krawędzi ekranu (patrz wiersz ❸).

Teraz po każdym kliknięciu przycisku *Gra* ustawienia w grze zostaną prawidłowo wyzerowane, co pozwoli nam zagrać dowolną ilość razy!

## Dezaktywacja przycisku Gra

Jedyny problem z przyciskiem *Gra* polega na tym, że tworzący go prostokąt zdefiniowany na ekranie będzie reagował na zdarzenia kliknięcia nawet wtedy, kiedy przycisk będzie niewidoczny. Jeżeli w trakcie gry przypadkowo klikniesz myszą obszar, w którym wcześniej został wyświetlony przycisk, gra rozpocznie się od początku!

Musimy to naprawić. Gra powinna się uruchamiać jedynie wtedy, gdy wartością `game_active` jest `False`.

Plik alien\_invasion.py:

---

```
def _check_play_button(self, mouse_pos):
    """Rozpoczęcie nowej gry po kliknięciu przycisku Gra przez użytkownika."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos) ❶
    if button_clicked and not self.game_active: ❷
        # Wyzerowanie danych statystycznych gry.
        self.stats.reset_stats()
        --cięcie--
```

---

Opcja `button_clicked` przechowuje wartość `True` lub `False` (patrz wiersz ❶). Gra zostanie ponownie uruchomiona jedynie po kliknięciu przycisku *Gra* i tylko wtedy, gdy aktualnie jest nieaktywna (patrz wiersz ❷). W celu przetestowania tego zachowania uruchom nową grę i klikaj myszą w obszarze, w którym powinien być wyświetlany przycisk *Gra*. Jeżeli wszystko działa zgodnie z oczekiwaniemi, kliknięcie obszaru przycisku *Gra* w trakcie toczej się rozgrywki nie powinno mieć żadnego wpływu na przebieg gry.

## Ukrycie kurSORA myszy

Kursor myszy musi być widoczny na początku, aby umożliwić rozpoczęcie gry, a później jest już niepotrzebny. Wprowadźmy więc zmianę polegającą na ukryciu kurSORA myszy, gdy gra stanie się aktywna. Przedstawiony tutaj fragment kodu umieść na końcu konstrukcji `if` w metodzie `_check_play_button()`.

Plik alien\_invasion.py:

---

```
def _check_play_button(self, mouse_pos):
    """Rozpoczęcie nowej gry po kliknięciu przycisku Gra."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.game_active:
        --cięcie--
        # Ukrycie kurSORA myszy.
        pygame.mouse.set_visible(False)
```

---

Przekazanie wartości `False` metodzie `set_visible()` nakazuje Pygame ukrycie kursora, gdy zostanie umieszczony nad oknem gry.

Kursor pojawia się z powrotem po zakończeniu rozgrywki, aby gracz mógł ponownie nacisnąć przycisk *Gra* i rozpocząć nową grę. Poniżej przedstawiłem kod odpowiedzialny za ponowne wyświetlenie kursora myszy.

*Plik alien\_invasion.py:*

```
def _ship_hit(self):
    """Reakcja na uderzenie obcego w statek."""
    if self.stats.ships_left > 0:
        --cięcie--
    else:
        self.game_active = False
        pygame.mouse.set_visible(True)
```

Kursor myszy staje się ponownie widoczny, gdy gra przechodzi w stan nieaktywny, co ma miejsce w trakcie wywołania funkcji `ship_hit()`. Zadbanie o tego rodzaju szczegóły powoduje, że gra wydaje się bardziej profesjonalna i pozwala graczowi skoncentrować się na samej rozgrywce, a nie na ustalaniu sposobu działania interfejsu użytkownika.

## ZRÓB TO SAM

**14.1. Naciśnij G, aby rozpoczęć grę.** Ponieważ w grze *Inwazja obcych* statkiem kosmicznym gracz kieruje za pomocą klawiatury, najlepszym rozwiązaniem będzie uruchamianie gry również przez naciśnięcie klawisza. Dodaj kod pozwalający graczowi rozpoczęć grę po naciśnięciu klawisza *G*. Pomoce może być przeniesienie pewnego fragmentu kodu z metody `_check_play_button()` do metody `_start_game()`, która będzie mogła być wywoływana zarówno z `_check_play_button()`, jak i z `_check_keydown_events()`.

**14.2. Inna gra.** Przy prawej krawędzi ekranu utwórz prostokąt, który z dość dużą szybkością będzie poruszał się w górę oraz w dół. Kierowany przez gracza statek kosmiczny powinien pojawić się po lewej stronie ekranu. Gracz może kierować tym statkiem w górę oraz w dół, a także strzelać pociskami. Dodaj przycisk pozwalający na rozpoczęcie gry. Gdy gracz trzykrotnie nie trafi w ruchomy cel, gra powinna się zakończyć, a na ekranie ponownie powinien się pojawić przycisk rozpoczęjący grę. Kliknięcie przycisku *Gra* powinno umożliwić graczowi ponowne rozpoczęcie gry.

## Zmiana poziomu trudności

Aktualnie w grze jest tak, że po zestrzeleniu całej floty obcych gracz przechodzi na kolejny poziom, choć nie wiąże się to ze wzrostem trudności rozgrywki. Ożywimy nieco grę i uczynimy ją bardziej interesującą w ten sposób, że będziemy zwiększać szybkość akcji za każdym razem, gdy graczowi uda się uniknąć wszystkich obcych.

## Zmiana ustawień dotyczących szybkości

Najpierw musimy przeorganizować klasę `Settings`, aby pogrupować ustawienia gry na statyczne i dynamiczne. Musimy mieć pewność, że te ustawienia będą się zmieniać, kiedy wyzerujemy grę i rozpoczęmy nową rozgrywkę. Poniżej przedstawiłem kod metody `__init__()` w pliku `settings.py`.

Plik `settings.py`:

---

```
def __init__(self):
    """Inicjalizacja danych statycznych gry."""
    # Ustawienia dotyczące ekranu.
    self.screen_width = 1200
    self.screen_height = 800
    self.bg_color = (230, 230, 230)

    # Ustawienia dotyczące statku kosmicznego.
    self.ship_limit = 3

    # Ustawienia dotyczące pocisku.
    self.bullet_width = 3
    self.bullet_height = 15
    self.bullet_color = 60, 60, 60
    self.bullets_allowed = 3

    # Ustawienia dotyczące obcego.
    self.fleet_drop_speed = 10

    # Latwa zmiana szybkości gry.
    self.speedup_scale = 1.1 ❶

    self.initialize_dynamic_settings() ❷
```

---

Kontynuujemy inicjalizację ustawień, które pozostają stałe w metodzie `__init__()`. W wierszu ❶ dodaliśmy ustawienie `speedup_scale` pozwalające na kontrolowanie zmiany szybkości gry — wartość 2 spowoduje podwojenie szybkości rozgrywki za każdym razem, gdy gracz przejdzie na nowy poziom, natomiast wartość 1 oznacza zachowanie tej samej szybkości. Wartość rzędu 1.1 powoduje wystarczający wzrost szybkości gry, choć jeszcze nie na tyle, aby uniemożliwić rozgrywkę. Na końcu mamy wywołanie `initialize_dynamic_settings()` odpowiedzialne za inicjalizację wartości dla atrybutów, które muszą ulegać zmianie w trakcie gry (patrz wiersz ❷).

Poniżej przedstawiłem kod metody `initialize_dynamic_settings()`.

Plik `settings.py`:

---

```
def initialize_dynamic_settings(self):
    """Inicjalizacja ustawień, które ulegają zmianie w trakcie gry."""
    self.ship_speed = 1.5
    self.bullet_speed = 2.5
```

---

```
self.alien_speed = 1.0  
# Wartość fleet_direction wynosząca 1 oznacza prawo, natomiast -1 oznacza lewo.  
self.fleet_direction = 1
```

---

Powyższa metoda powoduje ustawienie wartości początkowych dla szybkości poruszania się statku, pocisku oraz obcych. Te szybkości będziemy zwiększać w trakcie postępu poczynionego przez gracza w trakcie rozgrywki i zerować po rozpoczęciu nowej gry. W przedstawionej metodzie znajduje się atrybut `fleet_direction`, aby obcy zawsze poruszali się odpowiednio na początku każdej nowej gry. Nie trzeba inkrementować wartości `fleet_drop_speed`, ponieważ gdy obcy będą szybciej poruszać się po ekranie, to jednocześnie flota będzie szybciej zbliżać się do statku kierowanego przez gracza i dolnej krawędzi ekranu.

Aby zwiększyć szybkość poruszania się statku, pocisków i obcych za każdym razem, gdy gracz przejdzie na wyższy poziom, używamy metody `increase_speed()`.

*Plik settings.py:*

---

```
def increase_speed(self):  
    """Zmiana ustawień dotyczących szybkości."""  
    self.ship_speed *= self.speedup_scale  
    self.bullet_speed *= self.speedup_scale  
    self.alien_speed *= self.speedup_scale
```

---

Aby zwiększyć szybkość wymienionych elementów gry, szybkość każdego z nich mnożymy przez wartość przechowywaną w `speedup_scale`.

Tempo gry zwiększamy, wywołując metodę `increase_speed()` z poziomu `_check_bullet_alien_collisions()`, gdy ostatni obcy we flocie zostanie zestrzelony, ale jeszcze zanim zostanie utworzona nowa flota.

*Plik alien\_invasion.py:*

---

```
def _check_bullet_alien_collisions(self):  
    --cięcie--  
    if not self.aliens:  
        # Usunięcie istniejących pocisków, przyśpieszenie gry i utworzenie nowej floty.  
        self.bullets.empty()  
        self._create_fleet()  
        self.settings.increase_speed()
```

---

Zmiana wartości ustawień szybkości `ship_speed`, `alien_speed` i `bullet_speed` jest wystarczająca do tego, aby przyspieszyć całą grę!

## Wyzerowanie szybkości

Wszystkim zmienionym ustawieniom szybkości musimy przywrócić wartości początkowe za każdym razem, gdy gracz rozpocznie nową grę. W przeciwnym razie zastosowane będą poprzednio używane ustawienia dotyczące szybkości rozgrywki.

Plik alien\_invasion.py:

```
def _check_play_button(self, mouse_pos):
    """Rozpoczęcie nowej gry po kliknięciu przycisku Gra przez użytkownika."""
    button_clicked = self.play_button.rect.collidepoint(mouse_pos)
    if button_clicked and not self.game_active:
        # Wyzerowanie ustawień dotyczących gry.
        self.settings.initialize_dynamic_settings()
        --cięcie--
```

W tym momencie rozgrywka w *Inwazji obcych* dostarcza znacznie więcej radości, a sam gra stała się bardziej wymagająca. Kiedy gracz zniszczy całą flotę obcych, szybkość gry nieco się zwiększa, a tym samym rozgrywka staje się trudniejsza. Jeżeli gra będzie zbyt szybko stawała się trudna, zmniejsz wartość `settings.speedup_scale`. Natomiast jeśli gra nadal pozostaje zbyt łatwa, wówczas nieco zwiększą wskazaną wartość. Odszukaj najlepszą wartość, która powoduje wzrost trudności rozgrywki w rozsądny przedziale czasu. Pierwsze kilka poziomów powinno być łatwych, kilka kolejnych nieco bardziej wymagających, choć nadal możliwych do przejścia. Dopiero kolejne poziomy powinny być bardzo trudne, prawie niemożliwe do przejścia.

## ZRÓB TO SAM

**14.3. Zmiana trudności innej gry.** Pracę rozpocznij od kodu utworzonego w ćwiczeniu 14.2. Prostokąt powinien zwiększać szybkość wraz z postępem w grze. Kiedy zostanie kliknięty przycisk *Gra*, szybkość poruszania się prostokąta powinna zostać przywrócona do stanu początkowego.

**14.4. Poziomy trudności.** Dla gry *Inwazja obcych* przygotuj kilka przycisków umożliwiających graczyowi wybór odpowiedniego poziomu na początku gry. Każdy przycisk powinien przypisywać odpowiednie wartości atrybutom klasy `Settings` niezbędnym do zdefiniowania różnych poziomów trudności.

# Punktacja

Przystępujemy teraz do implementacji systemu punktacji, aby w czasie rzeczywistym monitorować punkty zdobywane przez gracza, a także wyświetlać najlepszy wynik, aktualny poziom oraz liczbę statków, jakie pozostały do dyspozycji gracza.

Punktacja jest wartością statystyczną, więc dodajemy atrybut `score` do klasy `GameStats`.

Plik game\_stats.py:

```
class GameStats:
    --cięcie--
    def reset_stats(self):
```

```
"""
Inicjalizacja danych statystycznych, które mogą zmieniać się w trakcie gry.
"""

self.ships_left = self.settings.ship_limit
self.score = 0
```

---

Aby wyzerować punktację po każdym uruchomieniu nowej gry, wartość atrybutu `score` inicjalizujemy w metodzie `reset_stats()` zamiast w `__init__()`.

## Wyświetlanie punktacji

By móc wyświetlić punktację na ekranie, zaczynamy od przygotowania nowej klasy `Scoreboard`. Na razie ta klasa będzie wyświetlała jedynie bieżącą punktację, choć później wykorzystamy ją również do podawania najlepszego wyniku, aktualnego poziomu oraz liczby statków, jakie pozostały do dyspozycji gracza. Poniżej przedstawiłem pierwszą część klasy; umieść ją w pliku o nazwie `scoreboard.py`.

Plik `scoreboard.py`:

---

```
import pygame.font

class Scoreboard:
    """Klasa przeznaczona do przedstawiania informacji o punktacji."""

    def __init__(self, ai_game):
        """Inicjalizacja atrybutów dotyczących punktacji."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.stats

        # Ustawienia czcionki dla informacji dotyczących punktacji.
        self.text_color = (30, 30, 30)
        self.font = pygame.font.SysFont(None, 48)

        # Przygotowanie początkowych obrazów z punktacją.
        self.prep_score()
```

---

Ponieważ klasa `Scoreboard` wyświetla tekst na ekranie, pracę rozpoczynamy od importowania modułu `pygame.font`. Następnie metodzie `__init__()` przekazujemy parametr `ai_game`, co pozwoli uzyskać dostęp do obiektów `settings`, `screen` i `stats`, aby można było wyświetlać monitorowane wartości (patrz wiersz ①). Kolejne kroki to zdefiniowanie koloru tekstu (patrz wiersz ②) oraz utworzenie obiektu czcionki (patrz wiersz ③).

Aby przekształcić w obraz tekst przeznaczony do wyświetlenia, w wierszu ④ wywołujemy metodę `prep_score()`, której kod przedstawiłem poniżej.

*Plik scoreboard.py:*

---

```
def prep_score(self):
    """Przekształcenie punktacji na wygenerowany obraz."""
    score_str = str(self.stats.score) ❶
    self.score_image = self.font.render(score_str, True, ❷
                                         self.text_color, self.settings.bg_color)

    # Wyświetlenie punktacji w prawym górnym rogu ekranu.
    self.score_rect = self.score_image.get_rect() ❸
    self.score_rect.right = self.screen_rect.right - 20 ❹
    self.score_rect.top = 20 ❺
```

---

W metodzie `prep_score()` zaczynamy od przekształcenia wartości liczbowej w `stats.score` na postać ciągu tekstowego (patrz wiersz ❶). Otrzymany ciąg tekstowy przekazujemy metodzie `render()`, która tworzy obraz (patrz wiersz ❷). Aby punktacja była wyraźnie widoczna na ekranie, metodzie `render()` przekazujemy kolor tła ekranu, a także kolor tekstu.

Obraz z punkcją zostaje umieszczony w prawym górnym rogu ekranu i będzie rozszerzany w lewą stronę wraz ze zwiększeniem się punktacji i szerokości przedstawiającej ją liczby. Aby mieć pewność o dosunięciu punktacji zawsze do prawej krawędzi ekranu, tworzymy prostokąt o nazwie `score_rect` (patrz wiersz ❸) i ustawiamy jego prawą krawędź w odległości 20 pikseli w lewo od prawej krawędzi ekranu (patrz wiersz ❹). Góra krawędź nowego prostokąta zostaje zdefiniowana w odległości 20 pikseli w dół od górnej krawędzi ekranu (patrz wiersz ❺).

Na końcu przygotowujemy metodę `show_score()` odpowiedzialną za wyświetlenie wygenerowanego obrazu z punkcją.

*Plik scoreboard.py:*

---

```
def show_score(self):
    """Wyświetlenie punktacji na ekranie."""
    self.screen.blit(self.score_image, self.score_rect)
```

---

Ta metoda wyświetla na ekranie, w położeniu wskazanym przez `score_rect`, obraz przedstawiający aktualną punktację.

## Utworzenie tablicy wyników

Wyświetlenie punktacji wymaga utworzenia w klasie `AlienInvasion` egzemplarza klasy `Scoreboard`. Zaczniż od uaktualnienia sekcji zawierającej polecenia `import`.

*Plik alien\_invasion.py:*

---

```
--cięcie--
from game_stats import GameStats
from scoreboard import Scoreboard
--cięcie--
```

---

Następnym krokiem jest utworzenie w metodzie `_init_()` egzemplarza klasy `Scoreboard`.

*Plik alien\_invasion.py:*

---

```
def __init__(self):
    --cięcie--
    pygame.display.set_caption("Inwazja obcych")

    # Utworzenie egzemplarza przeznaczonego do przechowywania danych
    # statystycznych gry oraz utworzenie egzemplarza klasy Scoreboard.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    --cięcie--
```

---

Pozostało już tylko wyświetlenie punktacji na ekranie, co odbywa się z poziomu metody `_update_screen()`.

*Plik alien\_invasion.py:*

---

```
def _update_screen(self):
    --cięcie--
    self.aliens.draw(self.screen)

    # Wyświetlenie informacji o punktacji.
    self.sb.show_score()

    # Wyświetlenie przycisku tylko wtedy, gdy gra jest nieaktywna.
    --cięcie--
```

---

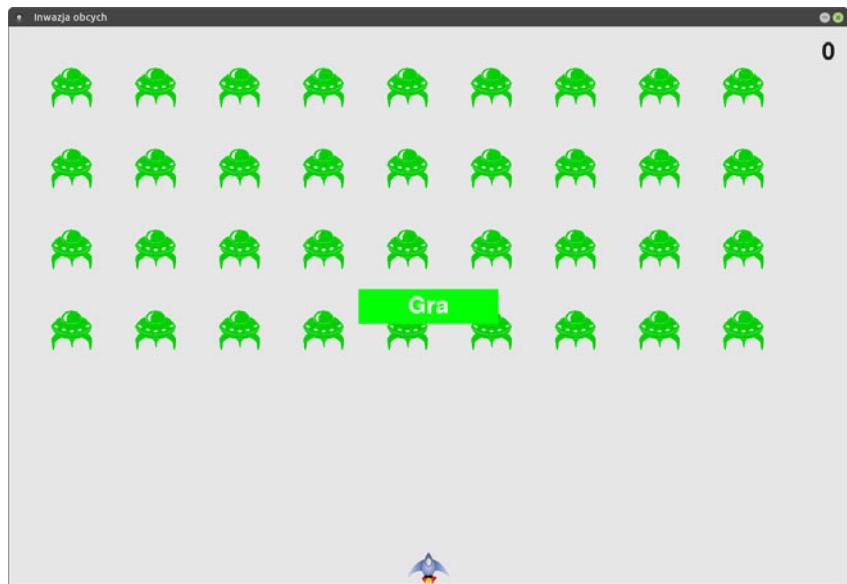
Metoda `show_score()` jest wywoływana tuż przed wyświetleniem przycisku *Gra* na ekranie.

Jeżeli teraz uruchomisz *Inwazję obcych*, powinieneś zobaczyć zero w prawym górnym rogu ekranu. (Na obecnym etapie prac chcemy się jedynie upewnić o wyświetleniu punktacji w odpowiednim miejscu na ekranie, a dopiero później zajmiemy się opracowaniem systemu punktacji). Na rysunku 14.2 pokazałem, jak jest wyświetlana punktacja tuż przed rozpoczęciem nowej gry.

Teraz przechodzimy do przypisania punktów poszczególnym obcym!

## **Uaktualnienie punktacji po zestrzeleniu obcego**

Aby przygotować uaktualnianą na bieżąco punktację, która będzie wyświetlana na ekranie, musimy zmieniać wartość atrybutu `stats.score` po każdym zestrzeleniu obcego, a następnie wywoływać funkcję `prep_score()`, by zaktualizować obraz wyświetlający punktację. Jednak w pierwszej kolejności musimy ustalić, ile punktów otrzyma gracz za zestrzelenie obcego.



Rysunek 14.2. Punktacja jest wyświetlona w prawym górnym rogu ekranu

Plik settings.py:

```
def initialize_dynamic_settings(self):
    --cięcie--
    # Punktacja.
    self.alien_points = 50
```

Liczba punktów będzie się zwiększała wraz z postępem poczynionym przez gracza. Aby mieć pewność, że wartość ta będzie się zerować za każdym razem po uruchomieniu nowej gry, będziemy ją ustawiać w `initialize_dynamic_settings()`.

Uaktualnienie liczby zdobytych punktów następuje w metodzie `_check_bullet_alien_collisions()` po każdym zestrzeleniu obcego przez gracza.

Plik alien\_invasion.py:

```
def _check_bullet_alien_collisions(self):
    """Reakcja na kolizję między pociskiem i obcym."""
    # Usunięcie wszystkich pocisków i obcych, między którymi doszło do kolizji.
    collisions = pygame.sprite.groupcollide(
        self.bullets, self.aliens, True, True)

    if collisions:
        self.stats.score += self.settings.alien_points
        self.sb.prep_score()
    --cięcie--
```

Kiedy pocisk trafia w obcego, Pygame zwraca słownik `collisions`. Sprawdzamy, czy ten słownik istnieje, a jeśli tak, liczbę punktów za trafienie danego obcego dodajemy do aktualnej liczby punktów. Następnie wywołujemy `prep_score()` w celu utworzenia nowego obrazu dla zaktualizowanej liczby zdobytych dotąd punktów.

Teraz, gdy zaczniesz grać w *Inwazję obcych*, powinieneś widzieć uaktualnianą na bieżąco liczbę zdobytych dotąd punktów.

## Zerowanie wyniku

Obecnie nowy wynik będzie przygotowywany jedynie po trafieniu obcego, co sprawdza się w większości sytuacji. Jednak po uruchomieniu nowej gry nadal będzie wyświetlane poprzedni wynik, aż do chwili pierwszego trafienia w obcego.

Rozwiązaniem tego problemu jest przygotowanie wyniku podczas uruchamiania nowej gry.

*Plik alien\_invasion.py:*

---

```
def _check_play_button(self, mouse_pos):
    --cięcie--
    if button_clicked and not self.game_active:
        --snip--
        # Wyzerowanie danych statystycznych gry.
        self.stats.reset_stats()
        self.sb.prep_score()
    --cięcie--
```

---

Metoda `prep_score()` jest wywoływana po wyzerowaniu danych statystycznych gry podczas uruchamiania nowej gry. W ten sposób wynik początkowy wynosi 0.

## Zagwarantowanie uwzględnienia wszystkich trafień

Aktualnie utworzony kod może nie uwzględniać pewnych trafień obcych. Na przykład jeśli dwa pociski będą mieć kolizję z obcymi w tej samej iteracji pętli lub utworzony przez nas wyjątkowo szeroki pocisk trafi jednocześnie wielu obcych, gracz otrzyma punkty tylko za jedno trafienie obcego. Aby usunąć ten problem, musimy dopracować sposób, w jaki wykrywane są kolizje między pociskami i obcymi.

W metodzie `_check_bullet_alien_collisions()` każdy pocisk trafiący w obcego staje się kluczem w słowniku `collisions`. Wartością przypisaną każdemu pociskowi jest lista obcych, z którymi miał kolizję. Przeprowadzamy iterację przez słownik `collisions` i upewniamy się, że punkty są dodawane za wszystkich trafionych obcych.

*Plik alien\_invasion.py:*

---

```
def _check_bullet_alien_collisions(self):
    --cięcie--
    if collisions:
        for aliens in collisions.values():
```

```
        self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
--cięcie--
```

---

Jeżeli słownik `collisions` został zdefiniowany, przeprowadzamy iterację przez wszystkie jego wartości. Pamiętaj, że każda wartość jest listą obcych trafionych przez pojedynczy pocisk. Wartość każdego obcego mnożymy przez liczbę obcych na każdej liście, a następnie dodajemy tę wartość do liczby zdobytych dotąd punktów. Aby przetestować to rozwiązanie, utwórz pocisk o szerokości 300 pikseli i sprawdź, czy otrzymasz punkty za trafienie każdego obcego takim superpociskiem. Nie zapomnij o przywróceniu pierwotnego wymiaru pocisku po zakończeniu testu.

## Zwiększenie liczby zdobywanych punktów

Ponieważ gra staje się coraz bardziej skomplikowana za każdym razem, gdy gracz przejdzie na kolejny poziom, za zestrzelenie obcego na wyższych poziomach gracz powinien otrzymywać większą liczbę punktów. Implementacja takiej funkcjonalności wymaga dodania kodu odpowiedzialnego za zwiększenie liczby punktów po przyśpieszeniu gry.

*Plik settings.py:*

---

```
class Settings:
    """ Klasa przeznaczona do przechowywania wszystkich ustawień gry."""

    def __init__(self):
        --cięcie--
        # Łatwa zmiana szybkości gry.
        self.speedup_scale = 1.1
        # Latwa zmiana liczby punktów przyznawanych za zestrzelenie obcego.
        self.score_scale = 1.5 ❶

        self.initialize_dynamic_settings()

    def initialize_dynamic_settings(self):
        --cięcie--

    def increase_speed(self):
        """Zmiana ustawień dotyczących szybkości gry i liczby przyznawanych
        punktów."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale

        self.alien_points = int(self.alien_points * self.score_scale) ❷
```

---

Definiujemy współczynnik używany podczas zwiększania liczby przyznawanych punktów i nadajemy mu nazwę `score_scale` (patrz wiersz ❶). Mały wzrost szybkości akcji w grze (1.1) powoduje, że poziom trudności gry wzrasta dość

szybko. Jednak zauważalna różnica w punktacji wymaga zmiany liczby przydzielanych punktów o nieco większy współczynnik, na przykład 1.5. Dlatego też kiedy zwiększymy tempo akcji w grze, musimy również zwiększyć liczbę punktów otrzymywanych przez gracza za każde trafienie obcego pociskiem (patrz wiersz ❷). Wykorzystujemy funkcję `int()` do zwiększania tej wartości o pełne liczby całkowite.

Aby poznać liczbę punktów otrzymywanych za unicestwienie poszczególnych obcych, dodaj wywołanie `print()` do metody `increase_speed()` w klasie `Settings`.

*Plik settings.py:*

---

```
def increase_speed(self):
    --cięcie--
    self.alien_points = int(self.alien_points * self.score_scale)
    print(self.alien_points)
```

---

Teraz, kiedy uruchomisz grę, powinieneś zobaczyć, że po każdym trafieniu obcego pociskiem, w terminalu pojawi się komunikat informujący o nowej wartości punktów. W tym momencie wartość zmiennej `alien_points` jest aktualizowana, ale nie jest nadal wykorzystywana w punktacji.

**UWAGA** *Nie zapomnij o usunięciu wymienionego wywołania `print()` po sprawdzeniu, czy liczba zdobywanych punktów faktycznie jest uaktualniana. W przeciwnym razie to wywołanie będzie miało negatywny wpływ na wydajność gry, a ponadto może niepotrzebnie rozpraszać gracza.*

## Zaokrąglanie punktacji

Większość gier zręcznościowych stosuje punktację będącą wielokrotnością liczby 10. Wprowadzimy więc odpowiednią zmianę w punktacji, aby zastosować się do tego kanonu. Sformatujmy punktację w taki sposób, aby był wyświetlany przecinek w charakterze separatora w dużych liczbach. Poniżej przedstawiłem zmiany, jakie należy wprowadzić w klasie `Scoreboard`.

*Plik scoreboard.py:*

---

```
def prep_score(self):
    """Przekształcenie punktacji na wygenerowany obraz."""
    rounded_score = round(self.stats.score, -1)
    score_str = f"{rounded_score:,}"
    self.score_image = self.font.render(score_str, True,
                                         self.text_color, self.settings.bg_color)
    --cięcie--
```

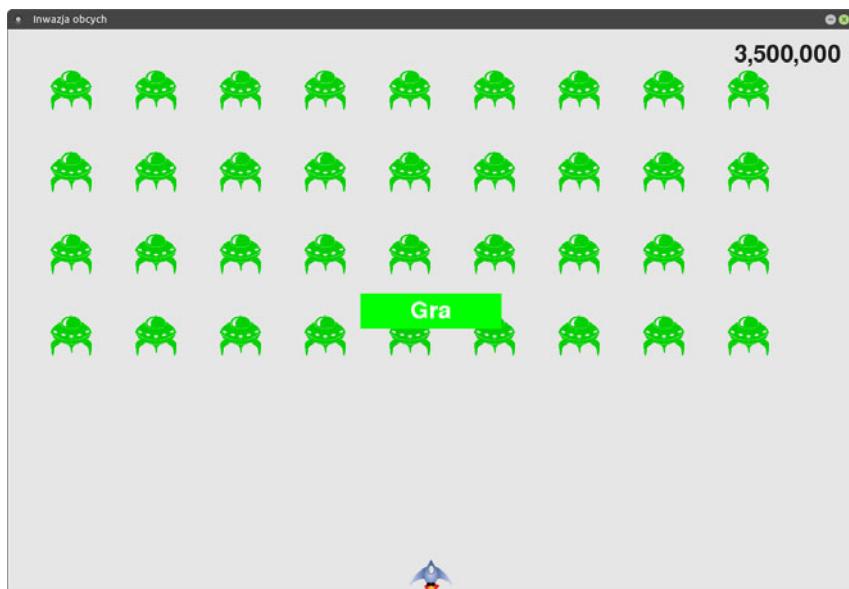
---

Działanie funkcji `round()` standardowo polega na zaokrąglaniu liczb zmienoprzecinkowych tak, aby zostało wyświetlone tyle cyfr po przecinku, ile zostało wskazane w wartości drugiego parametru. Jednak przekazanie wartości ujemnej jako drugiego parametru powoduje, że funkcja `round()` będzie zaokrągała wartość do najbliższej wielokrotności 10, 100, 1000 itd. Ten kod nakazuje Pythonowi

zaokrąglenie wartości stats.score do najbliższej 10 i umieszczenie wyniku w rounded\_score.

Następnie używamy specyfikatora formatu w tzw. ciągu tekstowym f. *Specyfikator formatu* to specjalna sekwencja znaków modyfikująca sposób, w jaki jest wyświetlana wartość zmiennej. W omawianym przykładzie sekwencja :, nakazuje Pythonowi wstawienie przecinków w liczbach podczas ich konwersji na ciąg tekstowy. Dlatego też wygenerowany będzie wynik 1,000,000 zamiast 1000000.

Teraz po uruchomieniu gry powinieneś otrzymać elegancko sformatowaną, zaokrągloną punktację, która pozostanie taka nawet wtedy, kiedy zdobędziesz wiele punktów, tak jak pokazałem na rysunku 14.3.



Rysunek 14.3. Zaokrąglona punktacja wraz z separatorami w postaci przecinków

## Najlepsze wyniki

Każdy gracz chce pobić najlepszy wynik osiągnięty dotąd w grze. Dlatego też będziemy monitorować najlepszy wynik, jaki kiedykolwiek został osiągnięty w grze, i wyświetlać go na ekranie. Najlepsze wyniki będą przechowywane za pomocą klasy GameStats.

Plik game\_stats.py:

```
def __init__(self, ai_game):
    --cięcie--
    # Najlepszy wynik nigdy nie powinien zostać wyzerowany.
    self.high_score = 0
```

Ponieważ najlepszy wynik nigdy nie powinien zostać wyzerowany, wartość `high_score` zainicjalizujemy w metodzie `__init__()`, a nie `reset_stats()`.

Musimy jeszcze zmodyfikować klasę `Scoreboard`, aby można było wyświetlić najlepszy wynik. Rozpoczynamy od metody `__init__()`.

*Plik scoreboard.py:*

```
def __init__(self, ai_game):
    --cięcie--
    # Przygotowanie początkowych obrazów z punktacją.
    self.prep_score()
    self.prep_high_score() ❶
```

Najlepszy wynik osiągnięty dotąd w grze nie będzie wyświetlany razem z aktualnym. Potrzebna jest więc nam nowa metoda o nazwie `prep_high_score()`, odpowiedzialna za przygotowanie obrazu przedstawiającego ten najlepszy wynik (patrz wiersz ❶).

Poniżej zaprezentowalem kod metody `prep_high_score()`.

*Plik scoreboard.py:*

```
def prep_high_score(self):
    """Konwersja najlepszego wyniku w grze na wygenerowany obraz."""
    high_score = round(self.stats.high_score, -1) ❶
    high_score_str = f"{high_score:,}"
    self.high_score_image = self.font.render(high_score_str, True, ❷
                                              self.text_color, self.settings.bg_color)

    # Wyświetlenie najlepszego wyniku w grze na środku ekranu, przy górnej krawędzi.
    self.high_score_rect = self.high_score_image.get_rect()
    self.high_score_rect.centerx = self.screen_rect.centerx ❸
    self.high_score_rect.top = self.score_rect.top ❹
```

Wynik zaokrąglamy do najbliższej wielokrotności liczby 10 i formatujemy do wyświetlenia wraz z przecinkami (patrz wiersz ❶). Następnie generujemy obraz na podstawie najlepszego wyniku uzyskanego dotąd w grze (patrz wiersz ❷). Prostokąt zawierający ten wynik wyrównujemy w poziomie (patrz wiersz ❸), a jego atrybutowi `top` przypisujemy wartość odpowiadającą atrybutowi `top` obrazu wyświetlającego bieżącą punktację (patrz wiersz ❹).

Metoda `show_score()` wyświetla teraz aktualną liczbę zdobytych punktów po prawej stronie ekranu, a najlepszy osiągnięty dotąd wynik na środku ekranu.

*Plik scoreboard.py:*

```
def show_score(self):
    """Wyświetlenie punktacji na ekranie."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
```

Aby wykonać sprawdzenie pod kątem najlepszego wyniku w grze, w klasie Scoreboard tworzymy nową metodę o nazwie `check_high_score()`.

Plik `scoreboard.py`:

---

```
def check_high_score(self):
    """Sprawdzenie, czy mamy nowy najlepszy wynik osiągnięty dotąd w grze."""
    if self.stats.score > self.stats.high_score:
        self.stats.high_score = self.stats.score
        self.prep_high_score()
```

---

Metoda `check_high_score()` jest używana do porównania aktualnej liczby punktów z najlepszym dotąd wynikiem w grze. Jeżeli bieżący wynik jest większy od dotąd najlepszego, uaktualniamy wartość `high_score` i wywołujemy `prep_high_score()`, aby zmienić obraz przedstawiający najlepszy wynik osiągnięty w grze.

Wywołanie `check_high_score()` jest konieczne po każdym trafieniu obcego i po uaktualnieniu wyniku w `_check_bullet_alien_collisions()`.

Plik `alien_invasion.py`:

---

```
def _check_bullet_alien_collisions(self):
    --cięcie--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
        self.sb.check_high_score()
    --cięcie--
```

---

Metodę `check_high_score()` wywołujemy, gdy istnieje słownik `collisions`. Wywołanie to następuje po tym, jak zostaje uaktualniona punktacja za wszystkich trafionych obcych.

W trakcie pierwszej gry w *Invazję obcych* aktualny wynik będzie jednocześnie najlepszym osiągniętym dotąd rezultatem. Na ekranie zobaczysz wyświetcone dwie takie same liczby wskazujące na punktację. Jednak kiedy rozpoczniesz drugą grę, najlepszy osiągnięty dotąd wynik będzie wyświetlany na środku ekranu, natomiast aktualna liczba punktów po prawej stronie, tak jak pokazałem na rysunku 14.4.

## Wyświetlenie aktualnego poziomu gry

Aby wyświetlić aktualny poziom gry, na jakim znajduje się gracz, przede wszystkim musimy mieć atrybut w klasie `GameStats` przedstawiający bieżący poziom. Wyzerowanie poziomu na początku każdej gry odbywa się w metodzie `reset_stats()`.



Rysunek 14.4. Najlepszy osiągnięty dotąd wynik w grze jest wyświetlany na górze pośrodku ekranu

Plik game\_stats.py:

---

```
def reset_stats(self):
    """Inicjalizacja danych statystycznych, które mogą zmieniać się w trakcie gry."""
    self.ships_left = self.settings.ship_limit
    self.score = 0
    self.level = 1
```

---

Konieczne jest zmodyfikowanie klasy Scoreboard do wyświetlania bieżącego poziomu. Nową metodę o nazwie `prep_level()` wywołujemy z poziomu metody `__init__()`.

Plik scoreboard.py:

---

```
def __init__(self, ai_game):
    --cięcie--
    self.prep_high_score()
    self.prep_level()
```

---

Poniżej przedstawiłem kod metody `prep_level()`.

### Plik scoreboard.py:

---

```
def prep_level(self):
    """Konwersja numeru poziomu na wygenerowany obraz."""
    level_str = str(self.stats.level)
    self.level_image = self.font.render(level_str, True, ❶
                                         self.text_color, self.settings.bg_color)

    # Numer poziomu jest wyświetlany pod aktualną punktacją.
    self.level_rect = self.level_image.get_rect()
    self.level_rect.right = self.score_rect.right ❷
    self.level_rect.top = self.score_rect.bottom + 10 ❸
```

---

Metoda `prep_level()` tworzy obraz na podstawie wartości przechowywanej w `level_str` (patrz wiersz ❶), a następnie przypisuje atrybutowi `right` obrazu wartość odpowiadającą atrybutowi `right` punktacji (patrz wiersz ❷). Kolejnym krokiem jest przypisanie atrybutowi `top` położenia 10 pikseli poniżej dolnej krawędzi obrazu punktacji, aby tym samym pozostawić nieco miejsca między punktacją i numerem poziomu (patrz wiersz ❸).

Konieczne jest również uaktualnienie metody `show_score()`.

### Plik scoreboard.py:

---

```
def show_score(self):
    """Wyświetlenie na ekranie punktacji oraz statków."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
```

---

W powyższym kodzie dodaliśmy wiersz odpowiedzialny za wyświetlenie na ekranie obrazu podającego numer poziomu, na którym aktualnie znajduje się gracz.

W metodzie `_check_bullet_alien_collisions()` inkrementujemy wartość `stats.level` i uaktualniamy obraz poziomu.

### Plik alien\_invasion.py:

---

```
def _check_bullet_alien_collisions(self):
    --cięcie--
    if not self.aliens:
        # Usunięcie istniejących pocisków, przyśpieszenie gry i utworzenie nowej floty.
        self.bullets.empty()
        self._create_fleet()
        self.settings.increase_speed()

        # Inkrementacja numeru poziomu.
        self.stats.level += 1
        self.sb.prep_level()
```

---

Jeżeli cała flota obcych zostanie zniszczona, inkrementujemy wartość stats.level i wywołujemy prep\_level(), by upewnić się, że informacja o nowym poziomie gry jest prawidłowo wyświetlnana.

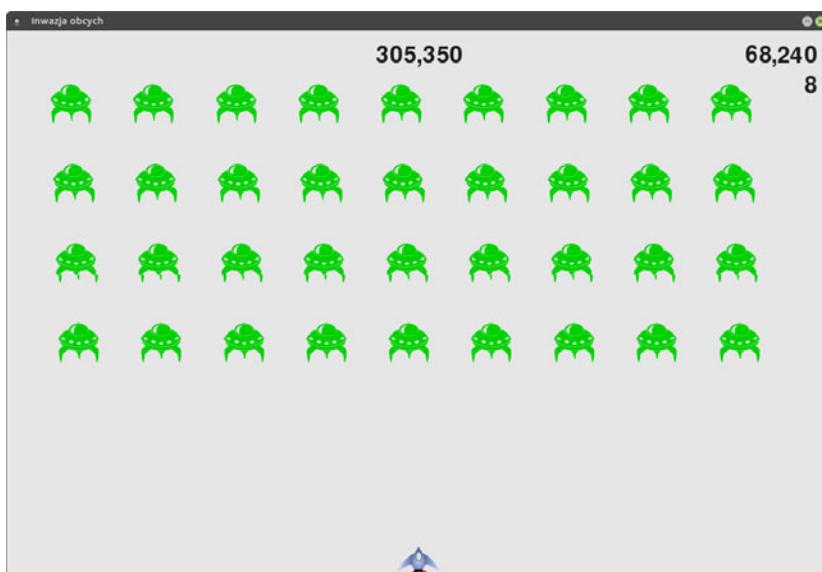
Aby mieć pewność, że obrazy przedstawiające bieżącą punktację i numer poziomu zostaną na początku nowej gry prawidłowo aktualnione, po kliknięciu przycisku *Gra* wywołujemy procedurę zerowania.

Plik alien\_invasion.py:

```
def _check_play_button(self, mouse_pos):
    --cięcie--
    if button_clicked and not self.game_active:
        --cięcie--
        self.sb.prep_score()
        self.sb.prep_level()
        --cięcie--
```

Metoda prep\_level() jest wywoływana tuż po prep\_score().

Kiedy wprowadzimy omówione powyżej modyfikacje, gracz zobaczy na ekranie aktualny poziom gry, tak jak pokazałem na rysunku 14.5.



Rysunek 14.5. Aktualny poziom gry jest wyświetlany tuż pod bieżącą punktacją

**UWAGA** W niektórych klasycznych grach punktacja zawiera także etykiety takie jak Punkty, Najlepszy wynik i Poziom. W budowanej tutaj grze pomineliśmy te etykiety, ponieważ znaczenie poszczególnych liczb staje się oczywiste po rozpoczęciu rozgrywki. Jeżeli jednak chcesz dodać wspomniane etykiety, umieść je przed ciągami tekstowymi przedstawiającymi liczby, a dokładnie przed wywołaniami font.render() w klasie Scoreboard.

## Wyświetlenie liczby statków

Na koniec wyświetlimy liczbę statków kosmicznych, jakie pozostały graczowi do dyspozycji, ale tym razem wykorzystamy do tego celu grafikę. Statki będą wyświetlane w lewym górnym rogu ekranu. Liczba statków odzwierciedla ich ilość pozostałą do dyspozycji gracza. Takie rozwiązanie jest podobne do stosowanego w wielu klasycznych grach zręcznościowych.

Konieczne będzie zdefiniowanie klasy `Ship` dziedziczącej po klasie `Sprite`, co pozwoli utworzyć grupę statków.

Plik `ship.py`:

---

```
import pygame
from pygame.sprite import Sprite

class Ship(Sprite):
    """Klasa przeznaczona do zarządzania statkiem kosmicznym."""

    def __init__(self, ai_game):
        """Inicjalizacja statku kosmicznego i jego położenie początkowe."""
        super().__init__() 2
        --cięcie--
```

---

Importujemy klasę `Sprite` i upewniamy się, że klasa `Ship` dziedziczy po klasie `Sprite` (patrz wiersz **1**). W wierszu **2** na początku metody `__init__()` wywołujemy metodę `super()`.

Następnym krokiem jest zmodyfikowanie klasy `Scoreboard` i utworzenie grupy statków, które będą wyświetlane. Poniżej przedstawiłem polecenia `import`.

Plik `scoreboard.py`:

---

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

---

Ponieważ tworzymy grupę statków, konieczne jest zainportowanie klas `Group` i `Ship`.

Spójrz na kod metody `__init__()`.

Plik `scoreboard.py`:

---

```
def __init__(self, ai_game):
    """Inicjalizacja atrybutów dotyczących punktacji."""
    self.ai_game = ai_game
    self.screen = ai_game.screen
    --cięcie--
    self.prep_level()
    self.prep_ships()
```

---

Egzemplarz przedstawiający grę przypisujemy atrybutowi, ponieważ konieczne będzie utworzenie statków. Metodę `prep_ships()` wywołujemy po `prep_level()`.

Poniżej przedstawiłem metodę `prep_ships()`.

*Plik scoreboard.py:*

```
def prep_ships(self):
    """Wyświetla liczbę statków, jakie pozostały graczyowi."""
    self.ships = Group() ❶
    for ship_number in range(self.stats.ships_left): ❷
        ship = Ship(self.ai_game)
        ship.rect.x = 10 + ship_number * ship.rect.width ❸
        ship.rect.y = 10 ❹
        self.ships.add(ship) ❺
```

Metoda `prep_ships()` tworzy pustą grupę o nazwie `self.ships` przeznaczoną do przechowywania egzemplarzy statku (patrz wiersz ❶). Wypełnienie tej grupy wymaga wykonania pętli dla każdego statku, jaki pozostał graczyowi do dyspozycji (patrz wiersz ❷). Wewnątrz pętli tworzymy nowy statek i przypisujemy mu wartość współrzędnej X, aby wszystkie statki pojawiły się w jednym rzędzie w odległości 10 pikseli od lewej krawędzi ekranu (patrz wiersz ❸). Wartość współrzędnej Y wskazuje położenie 10 pikseli poniżej górnej krawędzi ekranu — tym samym statki są wyrównane z obrazem aktualnej punktacji (patrz wiersz ❹). Na końcu dodajemy każdy nowy statek do grupy `ships` (patrz wiersz ❺).

Teraz konieczne jest wyświetlenie statków na ekranie.

*Plik scoreboard.py:*

```
def show_score(self):
    """Wyświetlenie na ekranie punktacji, poziomu oraz statków."""
    self.screen.blit(self.score_image, self.score_rect)
    self.screen.blit(self.high_score_image, self.high_score_rect)
    self.screen.blit(self.level_image, self.level_rect)
    self.ships.draw(self.screen)
```

W celu wyświetlania statków na ekranie wywołujemy metodę `draw()` dla grupy, a Pygame wyświetla wszystkie statki należące do tej grupy.

Aby gracz mógł widzieć liczbę statków, jakie pozostały mu do dyspozycji, na początku gry wywołujemy metodę `prep_ships()`. To wywołanie następuje z poziomu metody `_check_play_button()` zdefiniowanej w klasie `AlienInvasion`.

*Plik alien\_invasion.py:*

```
def _check_play_button(self, mouse_pos):
    --cięcie--
    if button_clicked and not self.game_active:
        --cięcie--
```

```
    self.sb.prep_level()
    self.sb.prep_ships()
--cięcie--
```

Metodę `prep_ships()` wywołujemy także po zderzeniu się statku kosmicznego z obcym, aby uwzględnić tę stratę statku przez gracza.

Plik `alien_invasion.py`:

```
def _ship_hit(self):
    """Reakcja na uderzenie obcego w statek."""
    if self.stats.ships_left > 0:
        # Zmniejszenie wartości przechowywanej w ships_left
        # i aktualnienie tablicy wyników.
        self.stats.ships_left -= 1
        self.sb.prep_ships()
--cięcie--
```

Wywołujemy `prep_ships()` po zmniejszeniu wartości `ships_left`, więc po każdej stracie statku wyświetlana jest prawidłowa liczba statków pozostalych graczy.

Na rysunku 14.6 możesz zobaczyć ukończony system punktacji oraz pozostałe graczy statki, które są wyświetlane po lewej stronie ekranu.



Rysunek 14.6. Ukończony system punktacji w grze „Inwazja obcych”

## ZRÓB TO SAM

**14.5. Najlepszy wynik wszechczasów.** Najlepszy wynik jest zerowany za każdym razem, gdy gracz zamyka grę, a później ponownie ją uruchamia. Rozwiąż ten problem przez zapis najlepszego wyniku w pliku tuż przed wywołaniem `sys.exit()` i odczytanie tego wyniku podczasinicjalizacji najlepszego wyniku w klasie `GameStats`.

**14.6. Refaktoryzacja.** Wyszukaj funkcje i metody wykonujące więcej niż tylko jedno zadanie, a następnie przeprowadź refaktoryzację kodu, aby zapewnić mu dobrą organizację i efektywność działania. Na przykład kod metody `_check_bullet_alien_collisions()` odpowiedzialny za utworzenie nowej floty obcych po jej zniszczeniu przez gracza przenieś do nowej metody o nazwie `start_new_level()`. Ponadto cztery oddzielne wywołania metod w `__init__()` klasy `Scoreboard` przenieś do metody o nazwie `prep_images()`, co pozwoli na skrócenie i uproszczenie kodu `__init__()`. Ta metoda `prep_images()` może również pomóc w działaniu `_check_play_button()` lub `start_game()`, jeśli już przeprowadziłeś refaktoryzację `_check_play_button()`.

**UWAGA** *Zanim podejmiesz próbę refaktoryzacji projektu, zajrzyj do dodatku D, aby dowiedzieć się, jak można przywrócić projekt, jeśli podczas refaktoryzacji przykładowo wprowadzisz błędy.*

**14.7. Rozbudowa gry *Inwazja obcych*.** Zastanów się, w jaki sposób można jeszcze dalej rozbudować tę grę. Na przykład obcy mogą strzelać w kierunku statku kosmicznego kierowanego przez gracza lub mieć osłony, za którymi będą się chować przed statkiem kierowanym przez gracza. Wspomniane osłony mogą być zniszczone przez pocisk dowolnej ze stron. Inną możliwość to wykorzystanie modułu `pygame.mixer`, aby dodać do gry efekty dźwiękowe, takie jak eksplozje i dźwięki wystrzałów.

**14.8. Strzelanie na boki — ostateczna wersja gry.** Kontynuuj pracę nad tą grą i wykorzystaj przy tym wszystko to, czego nauczyłeś się podczas pracy nad projektem *Inwazja obcych*. Dodaj przycisk *Gra*, zaimplementuj zwiększenie szybkości rozgrywki na kolejnych poziomach oraz odpowiedni system punktacji. Upewnij się o refaktoryzacji kodu podczas pracy i szukaj możliwości dostosowania gry do własnych potrzeb, wykraczając poza to, co zostało przedstawione w tym rozdziale.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak utworzyć przycisk *Gra* pozwalający na rozpoczęcie nowej gry oraz jak wykrywać zdarzenia myszy, dzięki czemu możesz ukrywać kursor, gdy gra jest aktywna. Zdobytą wiedzę możesz wykorzystać do przygotowania innych przycisków w grze, na przykład przycisku *Pomoc*, którego kliknięcie wyświetli informacje o tym, w jaki sposób należy grać. Zobaczyłeś też, jak zmodyfikować szybkość gry wraz z postępem poczionionym przez gracza, jak zaimplementować progresywny system punktacji i jak wyświetlać informacje w formie tekstu lub grafiki.

# 15

## Generowanie danych



**WIZUALIZACJA DANYCH** OBEJMUJE POZNAWANIE DANYCH ZA POMOCĄ ICH WIZUALNYCH REPREZENTACJI I JEST BLISKO ZWIĄZANA Z PROCESEM ANALIZY DANYCH, W KTÓRYM UŻYWAMY KODU DO POZNAWANIA WZORCÓW

ORAZ powiązań istniejących w zbiorze danych. Ten zbiór danych może być małą listą liczb mieszczących się w jednym wierszu kodu lub zbiorom przechowującym wiele terabajtów danych i składającym się z różnego rodzaju informacji.

Przygotowanie ładnie wyglądających reprezentacji danych to znacznie więcej niż tylko wykonanie ślicznych rysunków. Prosta i atrakcyjna reprezentacja zbioru danych pomaga oglądającemu w zrozumieniu znaczenia tych danych. Zaczyna on dostrzegać wagę elementów i wzorce, których istnienia w ogóle się nie spodziewał.

Na szczęście do wizualizacji skomplikowanych danych nie jest potrzebny superkomputer. Dzięki efektywności Pythona nawet na laptopie można bardzo szybko eksplorować zbiory danych składające się z milionów pojedynczych punktów danych. Oczywiście punktami danych nie muszą być liczby. Mając opanowane podstawy przedstawione w części pierwszej książki, możesz przeprowadzać analizę również danych innych niż liczbowe.

Python jest wykorzystywany do wykonywania naprawdę dużych operacji na danych w dziedzinach takich jak genetyka, badania nad klimatem czy analiza polityczna i ekonomiczna. Naukowcy zajmujący się danymi opracowali w Pythonie imponującą ilość narzędzi przeznaczonych do analizowania danych i tworzenia wizualizacji — wiele z tych narzędzi jest dostępnych także dla Ciebie. Jednym z najpopularniejszych narzędzi tego rodzaju jest matplotlib, czyli biblioteka matematyczna przeznaczona do tworzenia wykresów. Zastosujemy tę bibliotekę do

wygenerowania prostych wykresów, na przykład liniowego i punktowego. Następnie przejdziemy do bardziej interesujących zbiorów danych opartych na koncepcji błądzenia losowego (*random walk*), czyli wizualizacji wygenerowanej na podstawie serii losowych decyzji.

W tym projekcie użyjemy także pakietu o nazwie `plotly`, za pomocą którego można tworzyć wizualizacje działające doskonale na urządzeniach elektronicznych, np. do przeanalizowania wyników rzucania kością do gry. Pakiet generuje wizualizacje, które automatycznie zmieniają wielkość, aby dostosować się do różnych rozmiarów ekranów w urządzeniach. Wizualizacje te mogą zawierać także wiele interaktywnych funkcji, które można zastosować, by uwypuklić określone aspekty zbiorów danych, gdy użytkownik będzie umieszczał kurSOR myszy na poszczególnych fragmentach wykresu. Poznanie `matplotlib` i `plotly` pomoże rozpoczęć tworzenie wizualizacji dowolnego rodzaju danych, które Cię interesują.

## Instalacja matplotlib

Przede wszystkim konieczne jest zainstalowanie `matplotlib`, ponieważ tej biblioteki będziemy używać w pierwszych wizualizacjach. Do tego celu, podobnie jak w rozdziale 11. podczas instalowania `pytest`, wykorzystamy menedżer `pip` przeznaczony do pobierania i instalowania pakietów Pythona.

Aby zainstalować `matplotlib`, w powłoce należy wydać następujące polecenie:

---

```
$ python -m pip install --user matplotlib
```

---

Jeżeli do uruchomienia programów lub sesji Pythona w powłoce używasz innego polecenia niż `python`, np. `python3`, polecenie będzie miało następującą postać:

---

```
$ python3 -m pip install --user matplotlib
```

---

Jeśli chcesz zobaczyć, jakiego rodzaju wizualizacje można tworzyć za pomocą biblioteki `matplotlib`, odwiedź przykładową galerię w witrynie <https://matplotlib.org/> i kliknij łącze *Plot types*. Kiedy klikniesz na wizualizację w galerii, zostanie wyświetlony kod, który został użyty do wygenerowania danego wykresu.

## Wygenerowanie prostego wykresu liniowego

Przechodzimy do wygenerowania za pomocą biblioteki `matplotlib` prostego wykresu liniowego, który następnie zmodyfikujemy tak, aby przygotować wizualizację danych dostarczającą znacznie więcej informacji. Jako danych dla wykresu użyjemy sekwencji kwadratów kolejnych liczb, czyli 1, 4, 9, 16, 25.

Biblioteka matplotlib wystarczy po prostu podać sekwencję liczb, jak pokazaliem poniżej, a biblioteka zajmie się resztą.

Plik `mpl_squares.py`:

---

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
fig, ax = plt.subplots() ❶
ax.plot(squares)

plt.show()
```

---

Zaczynamy od zainportowania modułu `pyplot`, dla którego definiujemy alias `plt`, aby uniknąć konieczności nieustannego wpisywania `pyplot`. (Tę konwencję często będziesz spotykać w przykładach dostępnych w internecie i dlatego w tym miejscu również ją stosujemy). Moduł `pyplot` zawiera wiele różnych funkcji pomagających w generowaniu wykresów.

W kodzie przygotowaliśmy listę kwadratów kolejnych liczb, które mają zostać wyświetlane. Następnie została zastosowana kolejna konwencja matplotlib, polegająca na wywołaniu funkcji `subplots()` ❶. Ta funkcja pozwala wygenerować jeden lub więcej wykresów na tym samym rysunku. Zmienna `fig` przedstawia cały rysunek, czyli kolekcję wygenerowanych wykresów. Z kolei zmienna `ax` przedstawia jeden wykres na rysunku — jest to zmienna, z której najczęściej będziemy korzystać podczas definiowania pojedynczego wykresu i dostosowywania go do własnych potrzeb.

Kolejnym krokiem jest wywołanie metody `plot()`, która próbuje wyświetlić te liczby w jakiś logiczny sposób. Wywołanie `plt.show()` uruchamia dostarczaną wraz z matplotlib przeglądarkę i wyświetla w niej wykres, tak jak pokazałem na rysunku 15.1. Ta przeglądarka pozwala przybliżyć wykres i poruszać się po nim. Gdy klikniesz ikonę dyskietki, będziesz mógł zapisać dowolny obraz wykresu.

## Zmienianie etykiety i grubości wykresu

Mimo że wykres widoczny na rysunku 15.1 pokazuje wzrost liczb, to etykiety są zbyt małe, a sama linia zbyt cienka. Na szczęście matplotlib pozwala na dostosowanie każdej funkcji wizualizacji danych.

Wykorzystamy kilka dostępnych możliwości dostosowania wizualizacji do własnych potrzeb, aby wykres stał się czytelniejszy. Zaczniemy od dodania tytułu i etykiet dla obu osi.

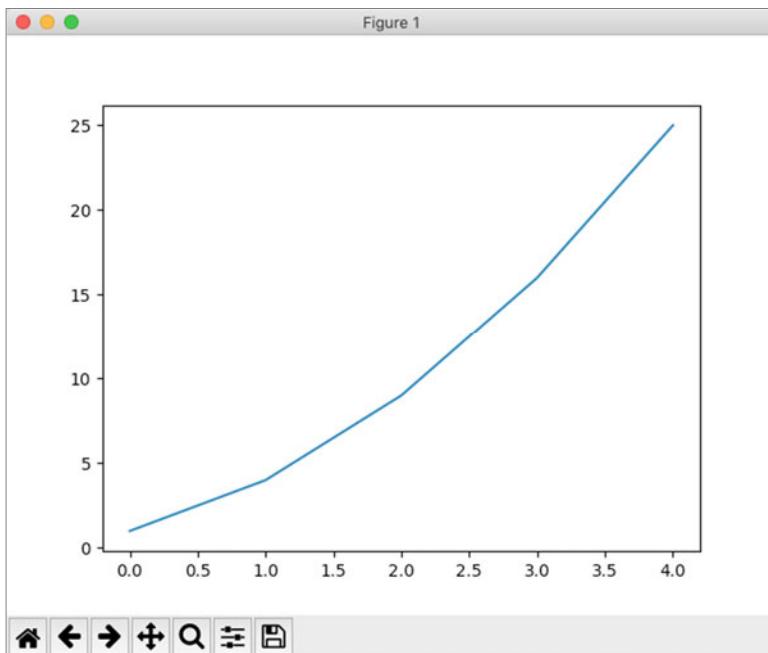
Plik `mpl_squares.py`:

---

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
```

---



Rysunek 15.1. Jeden z najprostszych wykresów, jaki można wygenerować za pomocą matplotlib

```
fig, ax = plt.subplots()
ax.plot(squares, linewidth=3) ❶

# Zdefiniowanie tytułu wykresu i etykiety osi.
ax.set_title("Kwadraty liczb", fontsize=24) ❷
ax.set_xlabel("Wartość", fontsize=14) ❸
ax.set_ylabel("Kwadraty wartości", fontsize=14)

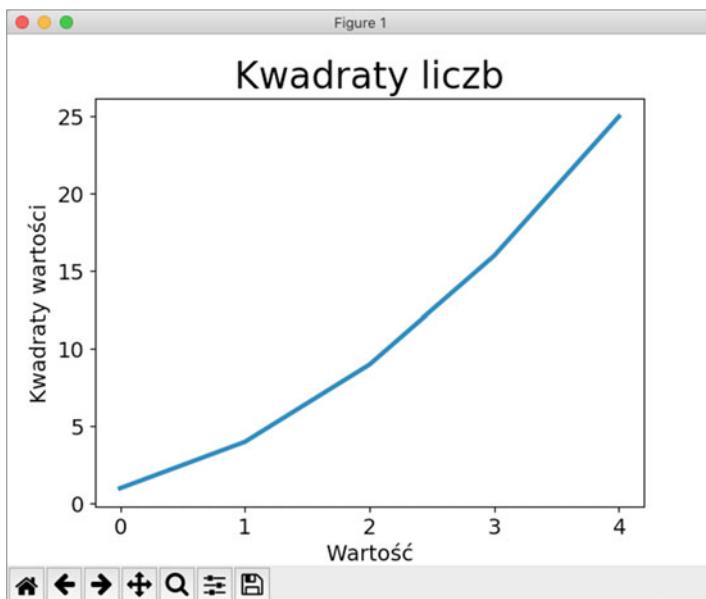
# Zdefiniowanie wielkości etykiet.
ax.tick_params(labelsize=14) ❹

plt.show()
```

Parametr `linewidth` widoczny w wierszu ❶ określa grubość linii generowanej przez wywołanie `plot()`. Po wygenerowaniu wykresu można skorzystać z wielu metod przeznaczonych do modyfikacji wykresu przed jego wyświetleniem. Metoda `set_title()` w wierszu ❷ definiuje tytuł wykresu. Parametr `fontsize` pojawiający się wielokrotnie w kodzie wskazuje wielkość tekstu na wykresie.

Metody `set_xlabel()` i `set_ylabel()` pozwalają zdefiniować tytuł dla każdej osi (patrz wiersz ❸), natomiast metoda `tick_params()` określa grubość linii ❹. Użyte w kodzie argumenty mają wpływ na grubość linii na osi X i Y (`axis="both"`) oraz określają wielkość etykiet znaczników na 14 (`labelsize=14`).

Jak możesz zobaczyć na rysunku 15.2, otrzymany wykres jest znacznie łatwiejszy do odczytania niż poprzedni. Etykiety są większe, a linia tworząca wykres grubsza. Warto poeksperymentować z tymi wartościami, aby sprawdzić, które z nich pozwalają wygenerować najlepiej wyglądający wykres.



Rysunek 15.2. Zmodyfikowany wykres jest znacznie łatwiejszy do odczytania

## Poprawianie wykresu

Skoro wykres stał się znacznie czytelniejszy, łatwiej można dostrzec istniejący w nim błąd, który polega na nieprawidłowym wyświetleniu danych. Zwróć uwagę, że na końcu wykresu jako kwadrat dla wartości 4.0 widnieje 25! Musimy to teraz poprawić.

Kiedy funkcji `plot()` przekazujemy sekwencję liczb, przyjmowane jest założenie, że pierwszy punkt danych odpowiada wartości 0 na osi X. Jednak w omawianym przykładzie pierwszy punkt danych odpowiada wartości 1 na osi X. Możemy nadpisać zachowanie domyślne przez przekazanie funkcji `plot()` wartości zarówno wyjściowych, jak i wejściowych, na podstawie których obliczyliśmy kwadraty liczb.

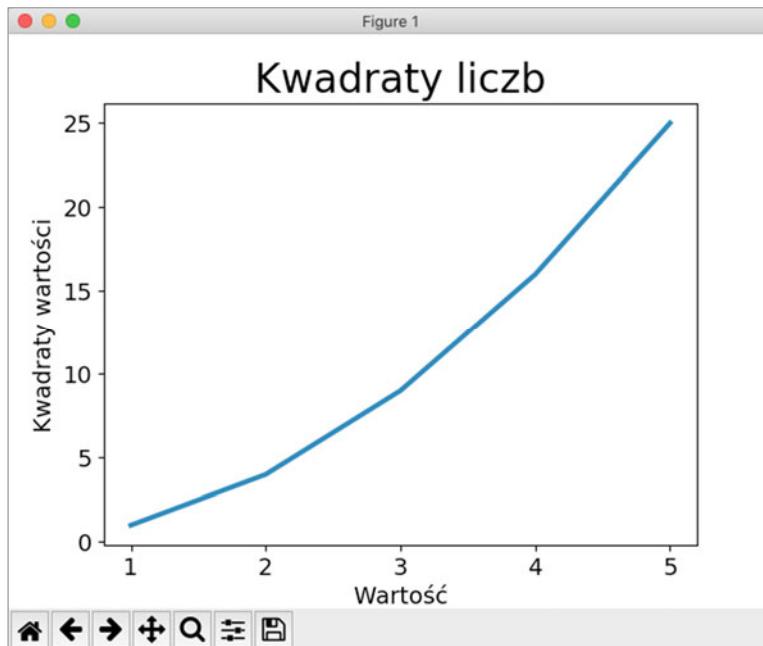
Plik `mpl_squares.py`:

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]
```

```
fig, ax = plt.subplots()  
ax.plot(input_values, squares, linewidth=3)  
# Zdefiniowanie tytułu wykresu i etykiet osi.  
--cięcie--
```

Teraz funkcja `plot()` wyświetla prawidłowy wykres, ponieważ podaliśmy jej wartości zarówno wejściowe, jak i wyjściowe, więc nie musi przyjmować żadnych założeń dotyczących sposobu wygenerowania liczb danych wyjściowych. Otrzymany w efekcie wykres (patrz rysunek 15.3) jest prawidłowy.



Rysunek 15.3. Dane na wykresie zostały teraz wyświetlone prawidłowo

Używając funkcji `plot()`, można stosować wiele parametrów oraz wykorzystywać wiele metod w celu dostosowania wykresu do własnych potrzeb. Te metody przeznaczone do dostosowywania wykresów będziemy poznawać w tym rozdziale podczas pracy ze znacznie bardziej interesującymi zbiorami danych.

## Używanie wbudowanych stylów

Matplotlib ma pewną liczbę predefiniowanych stylów, które zapewniają doskonałe ustawienia początkowe dla koloru tła, siatki, szerokości linii, rodzaju i wielkości czcionki itd. Dzięki tym stylom można bez większego wysiłku przygotowywać świetne wizualizacje. Aby poznać dostępne style, uruchom sesję Pythona w powłoce i wydaj następujące polecenia:

```
>>> import matplotlib.pyplot as plt  
>>> plt.style.available  
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery',  
--cięcie--
```

Jeżeli chcesz wykorzystać jeden z tych stylów, przed rozpoczęciem generowania wykresu musisz wstawić dodatkowy wiersz kodu.

Plik `mpl_squares.py`:

```
import matplotlib.pyplot as plt  
  
input_values = [1, 2, 3, 4, 5]  
squares = [1, 4, 9, 16, 25]  
  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
--cięcie--
```

Ten kod powoduje wygenerowanie wykresu pokazanego na rysunku 15.4. Do dyspozycji masz wiele różnych stylów, poeksperymentuj z nimi i znajdź te, które naprawdę Ci się podobają.



Rysunek 15.4. Wyświetlenie wykresu z zastosowaniem stylu `seaborn`

## Używanie funkcji scatter() do wyświetlania poszczególnych punktów i nadawania im stylu

Czasami użyteczne jest wyświetlenie poszczególnych punktów i nadanie im stylu na podstawie określonych cech charakterystycznych. Na przykład małe wartości będą wyświetlane w jednym kolorze, natomiast duże w drugim. Istnieje możliwość wyświetlenia dużego zbioru danych z użyciem jednego zestawu opcji stylu, a następnie podkreślenia wybranych punktów przez ich ponowne wyświetlenie z użyciem innych opcji.

Do wyświetlenia pojedynczego punktu można użyć funkcji `scatter()`. Kiedy przekażemy do funkcji wartości wskazujące pojedynczy punkt, na przykład `(x, y)`, funkcja `scatter()` powinna wyświetlić te wartości.

Plik `scatter_squares.py`:

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Przystępujemy do nadania stylu danym wyjściowym, aby były bardziej interesujące. Dodajemy tytuł, etykietę oraz osie i upewniamy się, że tekst jest wystarczająco łatwy do odczytania:

```
import matplotlib.pyplot as plt

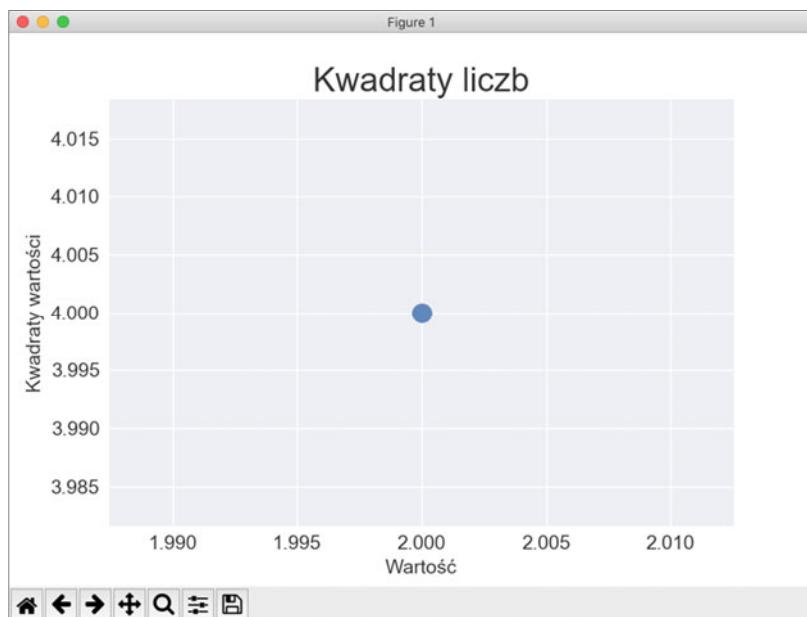
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4, s=200) ❶

# Zdefiniowanie tytułu wykresu i etykiety osi.
ax.set_title("Kwadraty liczb", fontsize=24)
ax.set_xlabel("Wartość", fontsize=14)
ax.set_ylabel("Kwadraty wartości", fontsize=14)

# Zdefiniowanie wielkości etykiety.
ax.tick_params(labelsize=14)

plt.show()
```

W wierszu ❶ wywołaliśmy funkcję `scatter()` i użyliśmy argumentu `s` w celu ustawnienia wielkości punktów, które będą wyświetlane na wykresie. Po uruchomieniu programu `scatter_squares.py` powinieneś zobaczyć pojedynczy punkt wyświetlony na środku wykresu, tak jak pokazałem na rysunku 15.5.



Rysunek 15.5. Wyświetlenie pojedynczego punktu

## Wyświetlanie serii punktów za pomocą funkcji scatter()

Aby wyświetlić serię punktów, funkcji scatter() można przekazać listę oddzielnych wartości X i Y, na przykład tak, jak pokazałem w poniższym programie.

Plik scatter\_squares.py:

---

```
import matplotlib.pyplot as plt

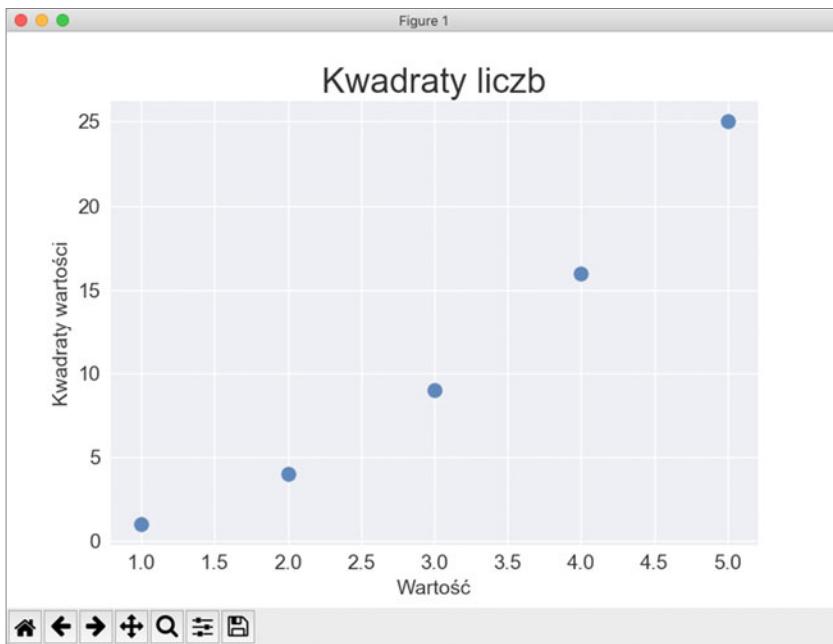
x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Zdefiniowanie tytułu wykresu i etykiet osi.
--cięcie--
```

---

Lista `x_values` zawiera liczby na podstawie których będą obliczone kwadraty, natomiast na liście `y_values` znajdują się już obliczone kwadraty. Kiedy te listy zostaną przekazane funkcji `scatter()`, biblioteka `matplotlib` odczyta po jednej wartości z każdej listy, a następnie wyświetli punkt. Wyświetlone zostaną punkty (1, 1), (2, 4), (3, 9), (4, 16) i (5, 25), a wygenerowany wykres możesz zobaczyć na rysunku 15.6.



Rysunek 15.6. Wykres punktowy zawierający wiele punktów

## Automatyczne obliczanie danych

Ręczne tworzenie przedstawionych list może być bardzo nieefektywne, zwłaszcza w przypadku wielu punktów. Zamiast przekazywać punkty na liście, lepiej zastosować pętlę i pozwolić Pythonowi na przeprowadzenie obliczeń.

Poniżej przedstawiłem kod przeznaczony do obliczenia 1000 punktów i ich wyświetlenia na wykresie.

Plik `scatter_squares.py`:

```
import matplotlib.pyplot as plt

x_values = range(1, 1001) ❶
y_values = [x**2 for x in x_values]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=10) ❷

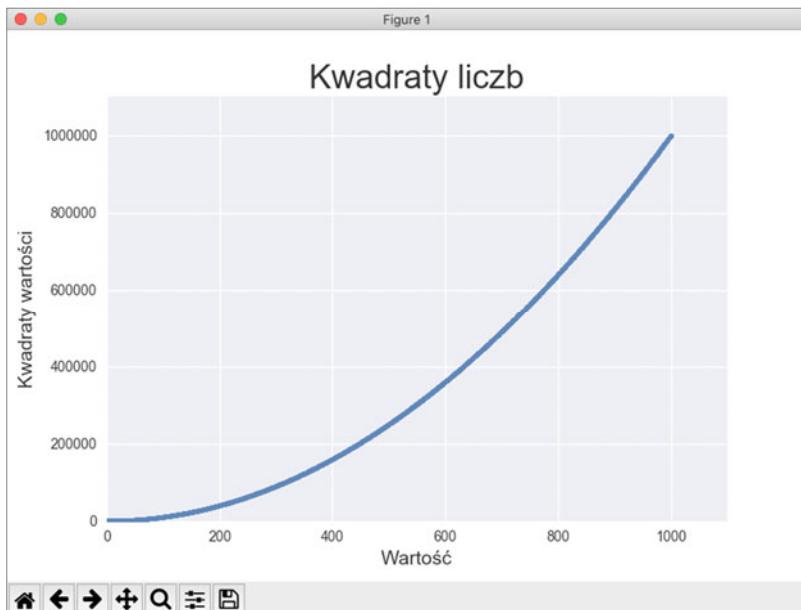
# Zdefiniowanie tytułu wykresu i etykiet osi.
--cięcie--

# Zdefiniowanie zakresu dla każdej osi.
ax.axis([0, 1100, 0, 1_100_000]) ❸

plt.show()
```

Rozpoczynamy od listy wartości X zawierającej liczby od 1 do 1000 (patrz wiersz ❶). W kolejnym kroku lista składana generuje wartości Y przez przeprowadzenie iteracji przez wartości X (`for x in x_values`), podniesienie każdej liczby do potęgi drugiej (`x**2`) i umieszczenie wyniku w `y_values`. W wierszu ❷ listy danych wejściowych i wyjściowych są przekazywane funkcji `scatter()`. Ponieważ to jest duży zbiór danych, używamy mniejszej wielkości punktu.

W celu określenia zakresu dla każdej osi używamy metody `axis()` (patrz wiersz ❸). Metoda `axis()` wymaga podania czterech wartości: wartości minimalnej i maksymalnej zarówno dla osi X, jak i Y. W omawianym fragmencie kodu dla osi X użyliśmy wartości od 0 do 1100, natomiast dla osi Y od 0 do 1 100 000. Wygenerowany na ich podstawie wykres pokazalem na rysunku 15.7.



Rysunek 15.7. Python może wyświetlić 1000 punktów równie łatwo jak 5 punktów

## Dostosowanie znaczników osi do własnych potrzeb

Gdy liczby wyświetlane na osiach są ogromne, matplotlib domyślnie będzie stosować notację naukową dla znaczników osi. Przeważnie jest to dobre rozwiązanie, ponieważ ogromne liczby zapisane w zwykłym formacie niepotrzebnie zajerają dużo miejsca na wizualizacji.

Praktycznie każdy element wykresu można dostosować do własnych potrzeb. Dlatego jeśli chcesz, możesz nakazać matplotlib, aby wartości dla osi były wyświetlane w zwykłym formacie.

---

--cięcie--

```
# Zdefiniowanie tytułu wykresu i etykiety osi  
ax.axis([0, 1100, 0, 1_100_000])  
ax.ticklabel_format(style='plain')  
  
plt.show()
```

---

Metoda `ticklabel_format()` pozwala nadpisać domyślny styl etykiet znaczników osi dla każdego wykresu.

## Definiowanie własnych kolorów

Aby zmienić kolor punktów, w wywołaniu funkcji `scatter()` przekaż parametr `color` wraz z nazwą koloru, którego chcesz użyć, tak jak pokazałem poniżej:

---

```
ax.scatter(x_values, y_values, color='red', s=10)
```

---

Istnieje również możliwość zdefiniowania własnego koloru za pomocą modelu RGB. W takim przypadku konieczne jest przekazanie argumentu `color` wraz z krotką zawierającą trzy wartości dziesiętne (po jednej dla koloru czerwonego, zielonego i niebieskiego) z zakresu od 0 do 1. Na przykład przedstawione poniżej wywołanie spowoduje utworzenie wykresu z jasnozielonymi punktami:

---

```
ax.scatter(x_values, y_values, color=(0, 0.8, 0), s=10)
```

---

Wartości bliższe 0 powodują wygenerowanie ciemnych kolorów, natomiast wartości bliższe 1 powodują wygenerowanie jaśniejszych kolorów.

## Użycie mapy kolorów

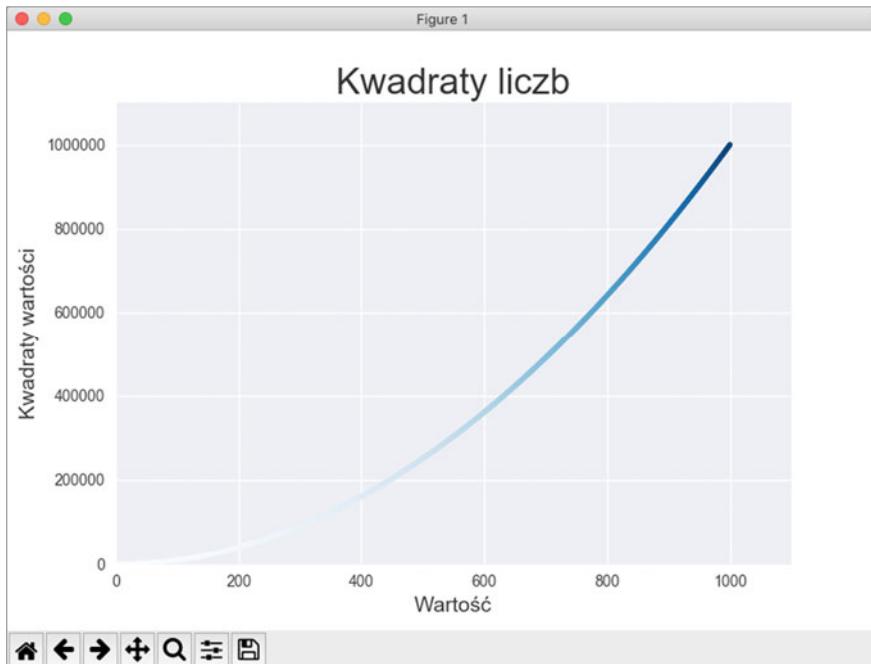
*Mapa kolorów* to seria kolorów w gradiencie przechodząca od koloru początkowego do końcowego. Mapy kolorów są używane w wizualizacjach do podkreślenia wzorca w danych. Na przykład niższe wartości mogą być przedstawiane za pomocą jaśniejszych kolorów, natomiast wyższe za pomocą ciemniejszych. Użycie map kolorów gwarantuje, że wszystkie punkty wizualizacji będą się zmieniać płynnie i dokładnie na dobrze opracowanej skali kolorów.

Moduł `pyplot` zawiera zestaw wbudowanych map kolorów. Aby użyć jednej z tych map kolorów, trzeba określić, jak moduł powinien przypisywać kolor poszczególnym punktom w zbiorze danych. W poniższym fragmencie kodu pokazałem przykład przypisywania punktowi koloru na podstawie wartości Y danego punktu.

Plik scatter\_squares.py:

```
--cięcie--  
plt.style.use('seaborn')  
fig, ax = plt.subplots()  
ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)  
  
# Zdefiniowanie tytułu wykresu i etykiety osi.  
--cięcie--
```

Argument `c` jest podobny do `color`, ale jest używany do powiązania sekwencji wartości z mapowaniem kolorów. Listę wartości Y przekazujemy argumentowi `c`, a następnie za pomocą argumentu `cmap` wskazujemy modułowi `pyplot` mapę kolorów przeznaczoną do użycia. W przedstawionym fragmencie kodu punkty o mniejszych wartościach Y mają kolor jasnoniebieski, natomiast punkty o większych wartościach Y mają kolor ciemnoniebieski. Wygenerowany na ich podstawie wykres pokazałem na rysunku 15.8.



Rysunek 15.8. Wykres wykorzystujący mapę kolorów o nazwie Blues

**UWAGA** Wszystkie mapy kolorów dostępne w module `pyplot` możesz zobaczyć w witrynie <http://matplotlib.org/>. Przejdz do samouczków (Tutorials), przewiń stronę w dół do sekcji Color, a następnie kliknij Choosing Colormaps in Matplotlib.

## Automatyczny zapis wykresu

Gdy chcesz, aby program automatycznie zapisywał wykres do pliku, wówczas wywołanie `plt.show()` możesz zastąpić wywołaniem `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

Pierwszy argument powyższego wywołania to nazwa pliku, w którym zostanie umieszczony wykres. Ten plik będzie zapisany w tym samym katalogu, w którym znajduje się program `scatter_squares.py`. Z kolei drugi argument powoduje usunięcie z wykresu dodatkowych białych znaków. Jeżeli jednak chcesz pozostawić dodatkowe miejsce wokół wykresu, możesz pominąć ten drugi argument. Istnieje również możliwość wywołania metody `savefig()` z obiektem typu `Path` i zapisańia pliku danych wyjściowych we wskazanym położeniu w systemie.

### ZRÓB TO SAM

**15.1. Sześciiany.** Liczba podniesiona do trzeciej potęgi jest *sześciianem*. Wygeneruj wykres dla sześciianów pierwszych pięciu liczb, a następnie przygotuj wykres dla sześciianów pierwszych 5000 liczb.

**15.2. Kolorowe sześciiany.** Zastosuj mapę kolorów na wykresach sześciianów.

## Błądzenie losowe

W tym podrozdziale wykorzystamy Pythona do wygenerowania danych dla błądzenia losowego, a następnie użyjemy matplotlib do utworzenia atrakcyjnej reprezentacji dla wygenerowanych danych. *Błądzenie losowe* to ścieżka pozbawiona wyraźnego kierunku, który jest ustalany na podstawie serii losowych decyzji, a ich wynik jest całkowicie nieprzewidywalny. Błądzenie losowe można porównać do drogi pokonywanej przez mrówkę, która się zgubiła i wykonuje każdy krok w zupełnie przypadkowo wybranym kierunku.

Błądzenie losowe jest wykorzystywane w praktycznych aplikacjach z takich dziedzin jak między innymi fizyka, biologia, chemia i ekonomia. Na przykład pyłki kwiatów na kropli wody poruszają się w różnych kierunkach po powierzchni tej kropli, ponieważ są nieustannie popchanie przez cząsteczki wody. Ruch cząsteczek w kropli wody jest całkowicie losowy, więc ścieżka pokonywana przez pyłki kwiatów po powierzchni jest błędzeniem losowym. Kod, który przygotujemy w tym podrozdziale, może służyć do modelowania wielu rzeczywistych sytuacji.

## Utworzenie klasy RandomWalk

W celu przeanalizowania błądzenia losowego utworzymy klasę `RandomWalk`, która będzie podejmować losowe decyzje dotyczące kierunku ruchu. Ta klasa wymaga podania trzech atrybutów: jednej zmiennej przeznaczonej do przechowywania

liczby punktów w błędzeniu oraz dwóch list przeznaczonych do przechowywania wartości współrzędnych X i Y każdego punktu w danym błędzeniu.

W klasie RandomWalk będziemy mieli jedynie dwie metody: `__init__()` i `fill_walk()` odpowiedzialną za obliczenie punktów dla danego błędzenia. Prace rozpocznamy od przedstawionej poniżej metody `__init__()`.

Plik `random_walk.py`:

---

```
from random import choice ①

class RandomWalk():
    """Klasa przeznaczona do wygenerowania błądzenia losowego."""

    def __init__(self, num_points=5000): ②
        """Inicjalizacja atrybutów błądzenia."""
        self.num_points = num_points

    # Punkt początkowy ma współrzędne (0, 0).
    self.x_values = [0] ③
    self.y_values = [0]
```

---

Aby umożliwić klasie podejmowanie losowych decyzji, wszystkie możliwe punkty przechowujemy na liście i używamy metody `choice()` do podjęcia decyzji w danej chwili (patrz wiersz ①). Następnie ustalamy domyślną liczbę punktów błędzenia losowego na 5000, czyli wystarczająco dużą, abyśmy mogli wygenerować interesujące wzorce, choć jednocześnie wystarczająco małą, aby dane były generowane szybko (patrz wiersz ②). W wierszu ③ tworzymy dwie listy przeznaczone do przechowywania wartości współrzędnych X i Y, a punkt początkowy określamy na (0, 0).

## Wybór kierunku

Do wygenerowania punktów i ustalenia kierunku każdego kroku wykorzystamy przedstawioną poniżej metodę `fill_walk()`. Umieść tę metodę w pliku `random_walk.py`.

Plik `random_walk.py`:

---

```
def fill_walk(self):
    """Wygenerowanie wszystkich punktów dla błądzenia losowego."""

    # Wykonywanie kroków aż do chwili osiągnięcia oczekiwanej liczby punktów.
    while len(self.x_values) < self.num_points: ①

        # Ustalenie kierunku oraz odległości do pokonania w tym kierunku.
        x_direction = choice([1, -1]) ②
        x_distance = choice([0, 1, 2, 3, 4])
        x_step = x_direction * x_distance ③
```

```

y_direction = choice([1, -1])
y_distance = choice([0, 1, 2, 3, 4])
y_step = y_direction * y_distance ❸

# Odrzucenie ruchów, które prowadzą donikąd.
if x_step == 0 and y_step == 0: ❹
    continue

# Ustalenie następnych wartości X i Y.
x = self.x_values[-1] + x_step ❺
y = self.y_values[-1] + y_step

self.x_values.append(x)
self.y_values.append(y)

```

---

W wierszu ❶ definiujemy pętlę działającą aż do chwili, gdy zostanie wygenerowana oczekiwana liczba punktów. Część główna tej metody wskazuje Pythonowi, jak mają być symulowane cztery losowe decyzje. Pierwsza służy do ustalenia kierunku w prawo lub w lewo. Druga określa odległość do pokonania w wybranym kierunku (w prawo lub w lewo). Trzecia ustala kierunek w górę lub w dół. Czwarta pozwala na określenie odległości do pokonania w ustalonym kierunku (w górę lub w dół).

Za pomocą wywołania `choice([1, -1])` wybieramy wartość dla `x_direction`. Wartością zwrotną będzie tutaj 1 dla ruchu w prawą stronę lub -1 dla ruchu w lewą stronę (patrz wiersz ❷). Następnie wywołanie `choice ([0, 1, 2, 3, 4])` wskazuje Pythonowi odległość do pokonania w tym kierunku (`x_distance`) przez losowy wybór liczby całkowitej z zakresu od 0 do 4. Uwzględnienie zera pozwala na wykonywanie kroków wzduż danej osi, jak również kroków określających poruszanie się wzduż obu osi.

W wierszach ❸ i ❹ określamy długość każdego kroku w kierunku X i Y, co odbywa się przez pomnożenie kierunku ruchu przez wybraną długość kroku. Wynik dodatni dla `x_step` oznacza ruch w prawą stronę, wynik ujemny oznacza ruch w lewą stronę, natomiast 0 oznacza ruch w pionie. Wartość dodatnia dla `y_step` oznacza ruch w góre, wartość ujemna oznacza ruch w dół, natomiast 0 oznacza ruch w poziomie. Jeżeli zarówno dla `x_step`, jak i `y_step` wartością jest 0, ruch zostaje zatrzymany, ale kontynuujemy pętlę, aby tego uniknąć (patrz wiersz ❺).

Aby pobrać następną wartość X dla ruchu, wartość w `x_step` dodajemy do ostatniej wartości przechowywanej w `x_values` (patrz wiersz ❻); to samo podejście stosujemy również względem wartości Y. Po zebraniu tych wartości dołączamy je do `x_values` i `y_values`.

## Wyświetlenie wykresu błądzenia losowego

Poniżej przedstawiłem kod przeznaczony do wyświetlenia wszystkich punktów wygenerowanych dla danego błądzenia losowego.

### Plik rw\_visual.py:

---

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Przygotowanie danych błądzenia losowego i wyświetlenie punktów.
rw = RandomWalk() ❶
rw.fill_walk()

# Wyświetlenie punktów błądzenia losowego.
plt.style.use('classic')
fig, ax = plt.subplots()
ax.scatter(rw.x_values, rw.y_values, s=15) ❷
ax.set_aspect('equal') ❸
plt.show()
```

---

Rozpoczynamy od zimportowania `pyplot` i `RandomWalk`. Następnie przygotowujemy nowe błądzenie losowe i umieszczamy je w zmiennej `rw` (patrz wiersz ❶) oraz upewniamy się, że została wywołana metoda `fill_walk()`. W wierszu ❷ metodzie `scatter()` przekazujemy wygenerowane wartości współrzędnych X i Y oraz wybieramy odpowiednią wielkość punktu. Domyślnie `matplotlib` niezależnie skaluje poszczególne osie. Jednak takie podejście spowoduje, że większość przykładów błądzenia losowego będzie rozciągnięta w poziomie lub pionie. Dlatego w omawianym przykładzie została użyta metoda `set_aspect()`, pozwalająca określić, że obie osie powinny mieć równe odległości między znacznikami osi ❸.

Na rysunku 15.9 pokazałem otrzymany wykres, który składa się z 5000 punktów. (Na rysunkach pokazywanych w tym podrozdziale nie widać interfejsu użytkownika przeglądarki `matplotlib`, który nadal będziesz widział po uruchomieniu programu `rw_visual.py`).

## Wygenerowanie wielu błądzeń losowych

Każde błądzenie losowe jest inne i sporo radości może dostarczyć poznawanie różnych wzorców, jakie mogą zostać wygenerowane. Jednym z możliwych rozwiązań jest opakowanie pętlą `while` przedstawionego wcześniej kodu i tym samym przygotowanie wielu błądzeń losowych bez konieczności wielokrotnego uruchamiania programu, na przykład tak, jak pokazałem poniżej.

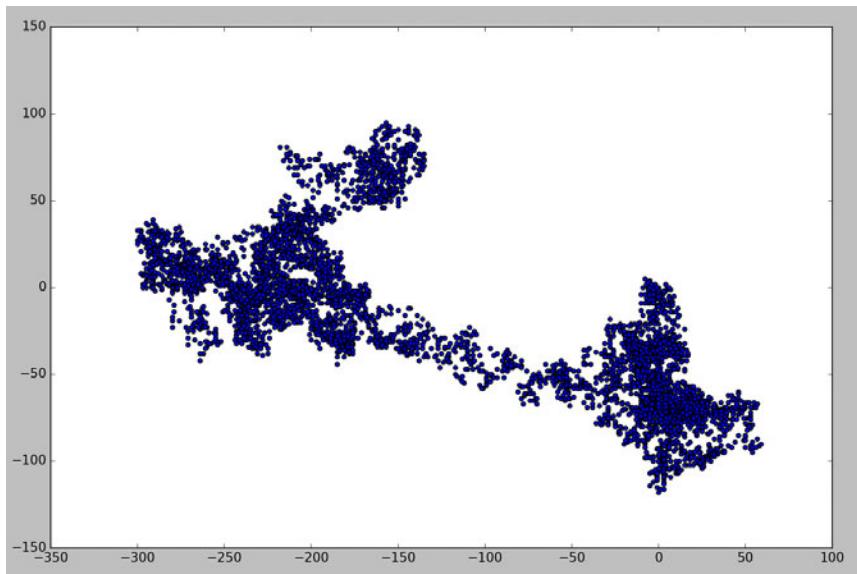
### Plik rw\_visual.py:

---

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Tworzenie nowego błądzenia losowego, dopóki program pozostaje aktywny.
while True:
    # Przygotowanie danych błądzenia losowego.
    --cięcie--
```



Rysunek 15.9. Wykres błądzenia losowego składający się z 5000 punktów

```
plt.show()
```

```
keep_running = input("Utworzyć kolejne błądzenie losowe? (t/n): ")
if keep_running == 'n':
    break
```

Ten kod spowoduje wygenerowanie błądzenia losowego, wyświetli je w przeglądarce matplotlib i będzie działał, dopóki przeglądarka pozostanie otwarta. Po jej zamknięciu zostanie wyświetcone pytanie, czy wygenerować następne błądzenie losowe. Odpowiedź twierdząca spowoduje wygenerowanie błądzenia losowego, które będzie pozostało w pobliżu zdefiniowanego punktu początkowego, będzie się oddalać głównie w jednym kierunku oraz będzie zawierać cienkie sekcje łączące większe grupy punktów. Kiedy będziesz chciał zakończyć działanie programu, naciśnij klawisz *n*.

## Nadawanie stylu danym wygenerowanym przez błądzenie losowe

W tej sekcji zmodyfikujemy wykresy, aby podkreślić ważne cechy charakterystyczne poszczególnych błądzeń losowych oraz nieco stonować rozpraszające nas elementy. W tym celu konieczne jest zidentyfikowanie cech charakterystycznych przeznaczonych do uwypuklenia, takich jak początek i koniec błądzenia losowego oraz zastosowana ścieżka. Następnie trzeba zidentyfikować cechy charakterystyczne przeznaczone do stonowania, takie jak znaczniki osi i etykiety. Wynikiem będzie prosta reprezentacja wizualna, która będzie wyraźnie pokazywać ścieżkę obraną w danym błądzeniu losowym.

## Kolorowanie punktów

Mapę kolorów wykorzystamy do pokazania kolejności punktów w błądzeniu losowym oraz usuniemy czarny kontur z każdego punktu, aby tym samym kolor poszczególnych punktów był wyraźnie widoczny. Aby punktom przypisać kolor zgodnie z ich położeniem w błądzeniu losowym, przekazujemy argumentowi `c` listę zawierającą położenie danego punktu. Ponieważ punkty są wyświetlane w kolejności, lista będzie zawierać numery od 0 do 4999, tak jak pokazałem w poniższym fragmencie kodu.

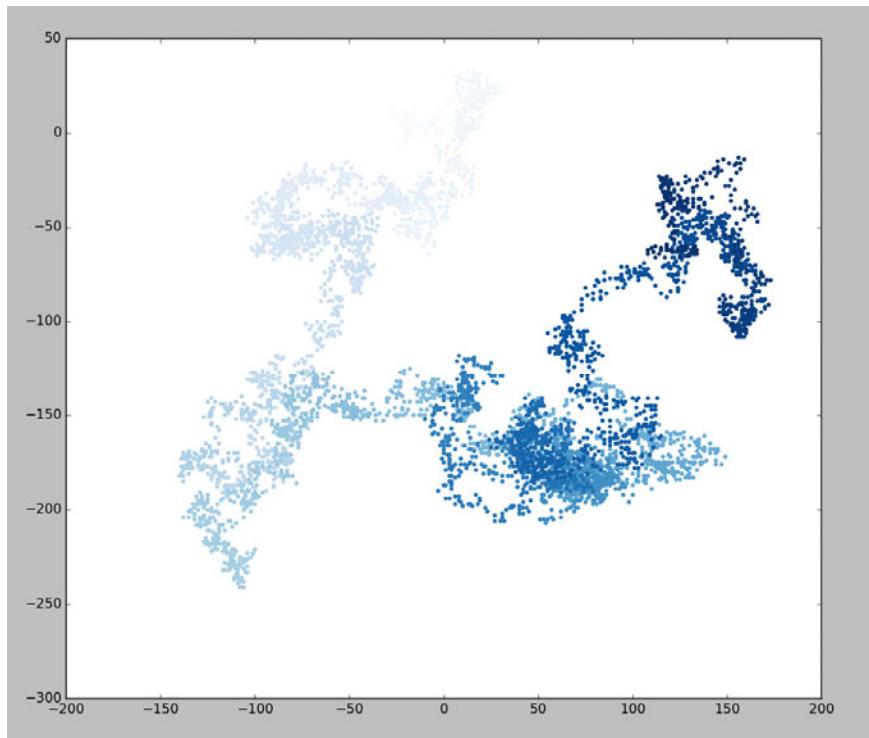
Plik `rw_visual.py`:

```
--cięcie--  
while True:  
    # Przygotowanie danych błądzenia losowego i wyświetlenie punktów.  
    rw = RandomWalk()  
    rw.fill_walk()  
  
    # Wyświetlenie punktów błądzenia losowego.  
    plt.style.use('classic')  
    fig, ax = plt.subplots()  
    point_numbers = range(rw.num_points) ❶  
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,  
               edgecolor='none', s=15)  
    ax.set_aspect('equal')  
    plt.show()  
--cięcie--
```

W wierszu ❶ wykorzystujemy funkcję `range()` do wygenerowania listy numerów, której wielkość odpowiada liczbie punktów w danym błądzeniu losowym. Te wygenerowane liczby będą przechowywane na liście `point_numbers`, której użyjemy do przypisania koloru poszczególnym punktom błądzenia losowego. Lista `point_numbers` zostanie przekazana argumentowi `c`. Ponadto użyjemy mapy kolorów `Blues` oraz argumentu `edgecolor=None` w celu pozbycia się konturu wokół każdego punktu. Wynikiem będzie wykres błądzenia losowego w kolorach od jasnoniebieskiego do ciemnoniebieskiego, dokładnie pokazujący błądzenie losowe od punktu początkowego do końcowego (patrz rysunek 15.10).

## Kolorowanie punktów początkowego i końcowego

Oprócz tego, że możemy pokolorować punkty w taki sposób, aby pokazać ich kolejność w błądzeniu losowym, dobrze byłoby też pokazać początek i koniec każdego błądzenia losowego. W tym celu pierwszy i ostatni punkt można wyświetlić oddzielnie, już po wyświetleniu całej serii punktów. Punkty początkowy i końcowy wyświetlimy jako większe oraz w zupełnie odmiennych kolorach, aby były doskonale widoczne na wykresie. Spójrz na poniższy fragmentu kodu.



Rysunek 15.10. Wykres błądzenia losowego wykorzystujący mapę kolorów Blues

Plik rw\_visual.py:

```
--cięcie--  
while True:  
    --cięcie--  
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers,  
               cmap=plt.cm.Blues, edgecolor='none', s=15)  
    ax.set_aspect('equal')  
  
    # Podkreślenie pierwszego i ostatniego punktu błądzenia losowego.  
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)  
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red',  
               edgecolors='none', s=100)  
  
    plt.show()  
--cięcie--
```

Aby wyświetlić punkt początkowy, definiujemy go w położeniu o współrzędnych (0, 0), w kolorze zielonym oraz w znacznie większym rozmiarze (s=100) niż pozostałe punkty. Z kolei punkt końcowy oznaczamy przez wygenerowanie ostatniej wartości X i Y w błądzeniu losowym oraz przypisujemy mu kolor czerwony

i wielkość 100. Upewnij się, że nowy kod wstawileś przed wywołaniem `plt.show()`, tak aby punkty początkowy i końcowy zostały wyświetcone nad wszystkimi pozostałymi punktami.

Kiedy uruchomisz ten kod, powinieneś od razu wyraźnie dostrzec początek i koniec danego błądzenia losowego. (Jeżeli te punkty nadal nie są dla Ciebie wyraźnie widoczne, zmień ich kolor i wielkość wedle własnego uznania).

## Ukrywanie osi

Ukryjemy teraz osie na wykresie, aby nie odciągały wzroku użytkownika od ścieżki błądzenia losowego. W celu ukrycia osi posłuż się przedstawionym poniżej kodem.

*Plik rw\_visual.py:*

```
--cięcie--  
while True:  
    --cięcie--  
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none', s=100)  
  
    # Ukrycie osi.  
    ax.get_xaxis().set_visible(False)  
    ax.get_yaxis().set_visible(False)  
  
    plt.show()  
--cięcie--
```

Do modyfikacji osi wykorzystujemy metody `ax.get_xaxis()` i `ax.get_yaxis()`, które pozwalają przypisać wartość `False` atrybutowi określającemu widoczność osi. W swojej dalszej pracy z wizualizacjami bardzo często będziesz się spotykać z tego rodzaju łączeniem metod.

Teraz uruchom `rw_visual.py` — powinieneś zobaczyć serię wykresów bez osi.

## Dodawanie punktów do wykresu

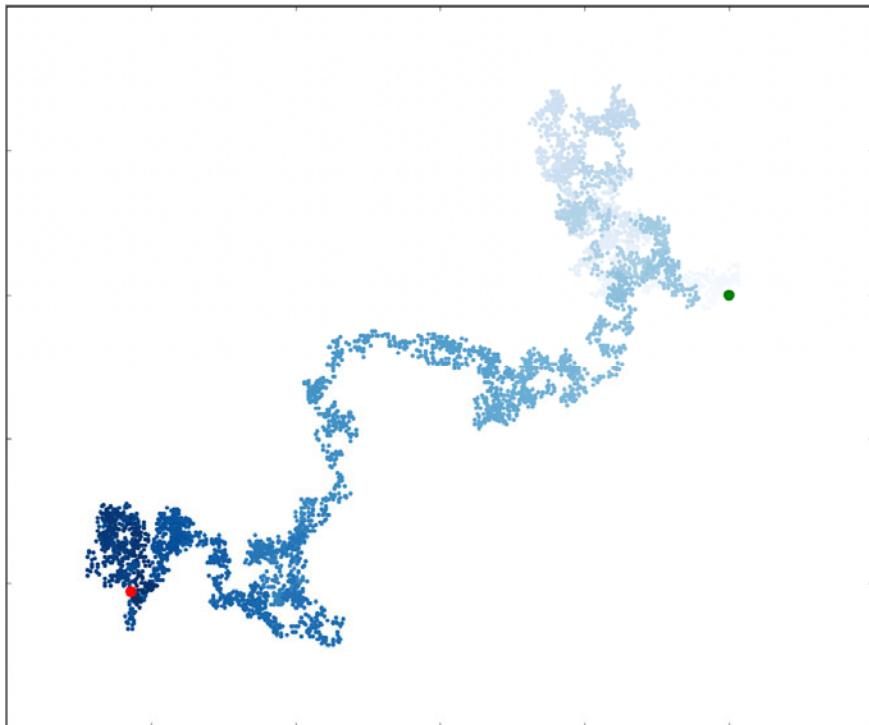
Zwiększymy teraz liczbę punktów, aby otrzymać większą ilość danych, z którymi można pracować. W tym celu musimy zmienić wartość `num_points` podczas tworzenia egzemplarza `RandomWalk` i dostosować wielkość każdego punktu w trakcie generowania wykresu, tak jak przedstawiłem w poniższym fragmencie kodu.

*Plik rw\_visual.py:*

```
--cięcie--  
while True:  
    # Przygotowanie danych błądzenia losowego i wyświetlenie punktów.  
    rw = RandomWalk(50_000)  
    rw.fill_walk()  
  
    # Wyświetlenie punktów błądzenia losowego.  
    plt.style.use('classic')  
    fig, ax = plt.subplots()
```

```
point_numbers = range(rw.num_points)
ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolor='none', s=1)
--cięcie--
```

W powyższym przykładzie tworzymy błądzenie losowe o wielkości 50 000 punktów (odzwierciedlenie rzeczywistych danych) i wyświetlamy każdy punkt o wielkości  $s=1$ . Otrzymane błądzenie losowe ma kształt strzepiasty i przypominający chmury (patrz rysunek 15.11). Jak możesz zobaczyć, na podstawie prostego wykresu punktowego przygotowaliśmy małe dzieło sztuki.



Rysunek 15.11. Wykres składający się z 50 000 punktów

Poeksperimentuj z tym kodem i zobacz, o ile możesz zwiększyć liczbę punktów w błądzeniu losowym, zanim system zacznie wyraźnie zwalniać lub wykres straci swoją wizualną atrakcyjność.

### Zmienianie wielkości wykresu, aby wypełnił ekran

Wizualizacja będzie znacznie efektywniej przedstawiać wzorce w danych, jeśli zostanie dopasowana do wielkości ekranu. Jeżeli chcesz okno wykresu lepiej dopasować do wielkości ekranu, rozwiązaniem jest zmiana wielkości okna danych wyjściowych matplotlib, na przykład w przedstawiony poniżej sposób.

---

```
fig, ax = plt.subplots(figsize=(15, 9))
```

---

Podczas tworzenia wykresu można przekazać argument `figsize` określający wielkość generowanego wykresu. Parametr `figsize` pobiera krotkę, która wskazuje bibliotecę matplotlib wyrażoną w calach wielkość okna wykresu.

Matplotlib przyjmuje, że rozdzielcość ekranu wynosi 100 pikseli na cal. Jeżeli kod nie powoduje wygenerowania wykresu o odpowiedniej wielkości, zmień wartości przekazywanych parametrów. Jeśli znasz rozdzielcość ekranu, przekaż ją funkcji `plt.subplots()` za pomocą parametru `dpi`. Dzięki temu możesz zdefiniować wielkość wykresu i jeszcze efektywniej wykorzystać dostępne miejsce na ekranie. Poniżej przedstawiłem przykład wywołania funkcji `plt.subplots()`:

---

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

---

To powinno pomóc w jak najefektywniejszym wykorzystaniu miejsca dostępnego na ekranie.

## ZRÓB TO SAM

**15.3. Zmodyfikuj ruch cząsteczek.** Zmodyfikuj `rw_visual.py` i zastąp wywołanie `ax.scatter()` wywołaniem `ax.plot()`. Aby zasymulować ścieżkę poruszania się pyłu kwiatów po powierzchni kropli wody, przekaż atrybuty `rw.x_values` i `rw.y_values` oraz dołącz atrybut `linewidth`. Zamiast 50 000 punktów użyj 5000.

**15.4. Zmodyfikowane błędzenie losowe.** W klasie `RandomWalk` wartości `x_step` i `y_step` są generowane na podstawie tego samego zestawu warunków. Kierunek jest wybierany losowo na podstawie listy `[1, -1]`, natomiast odległość na podstawie listy `[0, 1, 2, 3, 4]`. Zmodyfikuj wartości na tych listach i zobacz, jakie spowoduje to zmiany w kształcie generowanego wykresu. Spróbuj wydłużyć listę wyboru dla odległości, na przykład użyj wartości od 0 do 8, oraz usuń wartość `-1` z listy określającej kierunek następnego kroku.

**15.5. Refaktoryzacja.** Metoda `fill_walk()` jest długa. Utwórz nową metodę o nazwie `get_step()` przeznaczoną do ustalenia kierunku i odległości dla danego kroku, a następnie przygotowującą ten krok. Ostatecznie w `fill_walk()` powinny znaleźć się dwa wywołania `get_step()`, jak pokazałem poniżej:

---

```
x_step = self.get_step()
y_step = self.get_step()
```

---

Taka refaktoryzacja powinna zmniejszyć wielkość metody `fill_walk()`, która w ten sposób stanie się łatwiejsza do odczytania i zrozumienia.

# Symulacja rzutu kością do gry za pomocą plotly

W tym podrozdziale użyjemy opracowanego dla Pythona pakietu *plotly*, aby przygotować wizualizacje interaktywne. Pakiet jest szczególnie użyteczny podczas przygotowywania wizualizacji wyświetlanych w przeglądarkach WWW, ponieważ wizualizacje będą automatycznie skalowane do wypełnienia ekranu przeglądarki. Wizualizacje generowane przez *plotly* są również interaktywne — po umieszczeniu kurSORA myszy na wybranym elemencie na ekranie zostaną wyświetlane informacje dotyczące tego elementu. Początkowa wizualizacja zostanie utworzona za pomocą zaledwie kilku wierszy kodu z użyciem *plotly express*, czyli podzbioru *plotly* przeznaczonego do generowania wykresów na podstawie minimalnej ilości kodu. Po potwierdzeniu poprawności wykresu dane wyjściowe można dostosować do własnych potrzeb, podobnie jak w przypadku danych generowanych przez *matplotlib*.

W tym projekcie będziemy analizować wyniki rzutów kością do gry. W trakcie rzutu typową kością do gry masz jednakowe szanse na wyrzucenie dowolnej liczby od 1 do 6. Jednak kiedy rzucasz dwiema kościemi do gry, prawdopodobieństwo wyrzucenia pewnych liczb jest większe niż innych. Spróbujmy ustalić, które liczby prawdopodobnie wystąpią znacznie częściej niż inne. W tym celu wygenerujemy zbiór danych przedstawiający rzuty kością. Następnie wyświetlimy wyniki ogromnej liczby prób, aby tym samym ustalić, które wyniki najprawdopodobniej pojawią się częściej.

Wprawdzie przedstawione tutaj rozwiązanie pomaga modelować gry obejmujące rzuty kościemi, ale jego podstawowe koncepcje mają zastosowanie również w grach wykorzystujących dowolne aspekty losowości, np. w grach karcianych. Wiąże się również z wieloma rzeczywistymi sytuacjami, w których losowość odgrywa ważną rolę.

## Instalacja plotly

Zainstaluj *plotly* za pomocą menedżera *pip* podobnie jak w przypadku *matplotlib*.

---

```
$ python -m pip install --user plotly  
$ python -m pip install --user pandas
```

---

Moduł *plotly express* do działania potrzebuje *pandas*, czyli biblioteki przeznaczonej do efektywnej pracy z danymi. Dlatego ją również trzeba zainstalować. Jeżeli podczas instalacji *matplotlib* użyłeś polecenia *python3* lub innego, musisz odpowiednio zmodyfikować polecenie przedstawione powyżej.

Jeżeli chcesz zobaczyć, jakiego rodzaju wizualizacje można przygotować za pomocą *plotly*, zajrzyj do galerii na stronie <https://plot.ly/python/>. Każdy przykład zawiera kod źródłowy, więc będziesz mógł zobaczyć, jak dana wizualizacja została wygenerowana.

## Utworzenie klasy Die

Poniżej przedstawiłem klasę przeznaczoną do symulowania rzutu kością do gry.

*Plik die.py:*

---

```
from random import randint

class Die:
    """Klasa przedstawiająca pojedynczą kość do gry."""

    def __init__(self, num_sides=6): ❶
        """Przyjęcie założenia, że kość do gry ma postać sześciangu."""
        self.num_sides = num_sides

    def roll(self):
        """Zwrot wartości z zakresu od 1 do liczby ścianek, które ma kość
        →do gry."""
        return randint(1, self.num_sides) ❷
```

---

Metoda `__init__()` pobiera jeden opcjonalny argument (patrz wiersz ❶). W przypadku tej klasy tworzony egzemplarz kości zawsze będzie miał sześć ścian, o ile nie zostanie użyta inna wartość argumentu. Natomiast jeśli zostanie *podany* argument, jego wartość zostanie użyta jako liczba ścian kości do gry. Nazwa kości jest związana z liczbą ścianek i dlatego kość składająca się z sześciu ścianek to D6, natomiast z ośmiu ścianek to D8 itd.

Metoda `roll()` wykorzystuje funkcję `randint()` w celu zwrócenia losowej liczby z zakresu od 1 do liczby ścianek, które ma kość do gry (patrz wiersz ❷). Ta funkcja może zwrócić wartość początkową (1), wartość końcową (`num_sides`) lub dowolną liczbę całkowitą z zakresu definiowanego przez obie wymienione wartości.

## Rzut kością do gry

Zanim przystąpimy do utworzenia wizualizacji na podstawie przygotowanej klasy, najpierw rzucimy kością składającą się z sześciu ścian (D6), wyświetlimy wyniki i sprawdzimy, czy wyglądają sensownie.

*Plik die\_visual.py:*

---

```
from die import Die

# Utworzenie kości typu D6.
die = Die() ❶

# Wykonanie pewnej liczby rzutów i umieszczenie wyników na liście.
results = []
for roll_num in range(100): ❷
    result = die.roll()
    results.append(result)

print(results)
```

---

W wierszu ❶ tworzymy egzemplarz klasy `Die`, czyli kość do gry o domyślnie sześciu ściankach. Następnie w wierszu ❷ rzucamy nią stukrotnie i wynik każdego rzutu umieszczamy na liście `results`. Poniżej przedstawiłem przykładowy zbiór wyników.

---

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

---

Rzut okna na przedstawione wyniki pozwala przyjąć założenie, że klasa `Die` działa zgodnie z oczekiwaniemi. Otrzymujemy wartości 1 i 6, więc wiemy, że zwracane są wartość najmniejsza i największa. Ponieważ nie widzimy wartości 0 i 7 to wiemy, że wynikiem rzutu jest wartość z prawidłowo zdefiniowanego zakresu. Lista wyników zawiera wartości z zakresu od 1 do 6, czyli zawiera wszystkie liczby możliwe do otrzymania. Ustalimy teraz dokładną liczbę wystąpień poszczególnych wartości.

## Analiza wyników

Wykonajmy pewną liczbę rzutów jedną kością typu D6, a następnie przeanalizujmy wyniki, zliczając wystąpienia poszczególnych wartości.

Plik `die_visual.py`:

---

```
--cięcie--
# Wykonanie pewnej liczby rzutów i umieszczenie wyników na liście.
results = []
for roll_num in range(1000): ❶
    result = die.roll()
    results.append(result)

# Analiza wyników.
frequencies = []
poss_results = range(1, die.num_sides+1) ❷
for value in poss_results:
    frequency = results.count(value) ❸
    frequencies.append(frequency) ❹

print(frequencies)
```

---

Ponieważ nie ograniczamy się już do prostego wyświetlenia wyników, tylko do przeprowadzenia analizy, liczbę symulacji można zwiększyć do 1000 (patrz wiersz ❶). Na potrzeby analizy tworzymy pustą listę o nazwie `frequencies` przeznaczoną do przechowywania liczby wystąpień danej wartości. Następnie generujemy wszystkie możliwe wartości (w omawianym przykładzie to zakres od 1 do 6), czyli od 1 do liczby ścianek kości, jak pokazałem w wierszu ❷. Przeprowadzamy

iterację przez otrzymane wyniki i obliczamy liczbę wystąpień każdej wartości w resułts (patrz wiersz ❸), a następnie dołączamy tę wartość do listy frequencies (patrz wiersz ❹). Przed utworzeniem wizualizacji wyświetlamy przygotowaną listę częstotliwości występowania poszczególnych wartości:

---

```
[155, 167, 168, 170, 159, 181]
```

---

Te wyniki wyglądają sensownie. Mamy sześć liczb, po jednej dla każdej wartości możliwej do otrzymania po rzuceniu kością składającą się z sześciu ścianek. Żadna wartość nie jest szczególnie większa od pozostałych. Przechodzimy więc do wizualizacji wyników

## Utworzenie histogramu

Mając przygotowaną listę częstotliwości występowania poszczególnych wartości, możemy utworzyć histogram tych wyników. Do tego wystarczy zaledwie kilka wierszy kodu wykorzystującego plotly express.

Plik die\_visual.py:

---

```
import plotly.express as px

from die import Die
--cięcie--

for value in poss_results:
    frequency = results.count(value)
    frequencies.append(frequency)

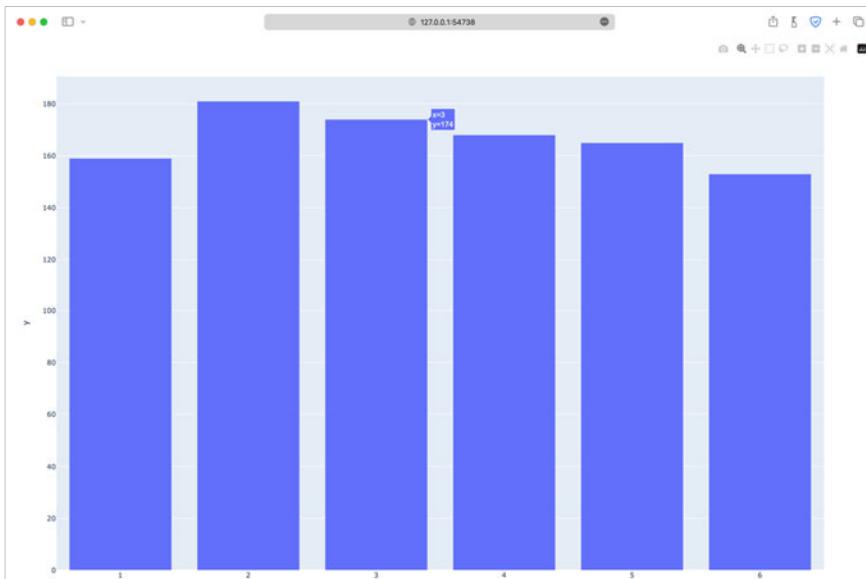
# Wizualizacja wyników.
fig = px.bar(x=poss_results, y=frequencies)
fig.show()
```

---

Zaczynamy od zimportowania modułu `plotly.express` i wykorzystania dla niego wygodnego aliasu `px`. Do utworzenia wykresu słupkowego zostanie użyta funkcja `px.bar()`. W najprostszym wywołaniu tej funkcji trzeba jej przekazać jedynie zbiory wartości `X` i `Y`. W omawianym przykładzie wartości `X` reprezentują możliwe wyniki rzutu kością, a wartości `Y` to częstotliwość występowania poszczególnych wyników.

W ostatnim wierszu kodu mamy wywołanie `fig.show()`, nakazujące Pythonowi wygenerowanie wykresu wynikowego jako pliku HTML i jego wyświetlenie w przeglądarce WWW. Ten wykres możesz zobaczyć na rysunku 15.12.

To naprawdę bardzo prosty wykres i zdecydowanie niepełny. Jednak w taki sposób ma być używany `plotly express` — tworzysz kilka wierszy kodu, patrzysz na wykres i upewniasz się, że przedstawia dane, które chcesz na nim umieścić. Jeżeli podoba Ci się to, co widzisz, możesz zająć się dostosowaniem elementów



Rysunek 15.12. Prosty wykres słupkowy wygenerowany przez `plotly express`

wykresu do własnych potrzeb, np. przez modyfikację etykiety i stylów. Jeżeli jednak chcesz poznać jeszcze inne dostępne typy stylów, nie musisz poświęcać na to wiele czasu. Skorzystaj teraz z tej możliwości przez zmianę wywołania `px.bar()` na innego, np. `px.scatter()` lub `px.line()`. Pełną listę dostępnych wykresów znajdziesz na stronie <https://plotly.com/python/plotly-express/>.

Ten wykres jest dynamiczny i interaktywny. Jeżeli zmienisz wielkość okna przeglądarki WWW, wielkość wykresu również ulegnie zmianie w celu jego dopasowania do dostępnej ilości miejsca. Po umieszczeniu kurSORA myszy nad dowolnym słupkiem zostaną wyświetlane powiązane z nim dane.

## Dostosowanie wykresu do własnych potrzeb

Po potwierdzeniu, że został użyty właściwy rodzaj wykresu, a przedstawione na nim dane są poprawne, można skoncentrować się na dodaniu odpowiednich etykiety i stylów dla wykresu.

Pierwszym krokiem podczas dostosowywania wykresu plotly do własnych potrzeb jest użycie parametrów opcjonalnych w wywołaniu początkowym odpowiedzialnym za wygenerowanie wykresu. W omawianym przykładzie jest to `px.bar()`. Spójrz na fragment kodu pokazujący, jak można dodać tytuły i etykiety do poszczególnych osi.

Plik `dice_visual.py`:

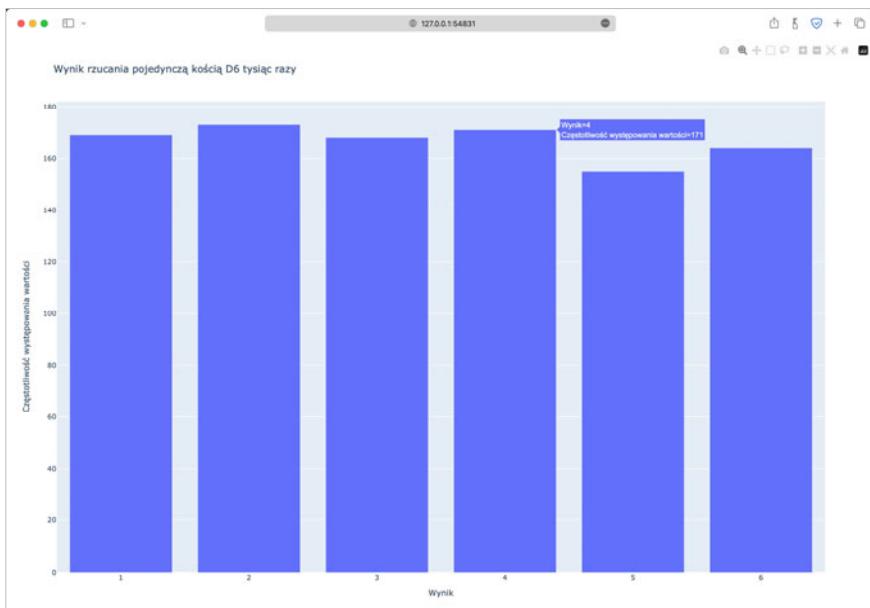
---

```
--cięcie--
# Wizualizacja wyników.
title = "Wynik rzucania pojedynczą kością D6 tysiąc razy" ❶
```

```
labels = {'x': 'Wynik', 'y': 'Częstotliwość występowania wartości'} ❷
fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)
fig.show()
```

Na początku został zdefiniowany żądany tytuł i przypisany zmiennej `title` (patrz wiersz ❶). W celu zdefiniowania etykiet osi konieczne jest użycie słownika (patrz wiersz ❷). Klucze słownika odwołują się do etykiet, które mają zostać zmodyfikowane, natomiast wartości to etykiety niestandardowe, które mają zostać użyte. W tym przykładzie oś X będzie zatytułowana *Wynik*, natomiast oś Y będzie miała tytuł *Częstotliwość występowania wartości*. Wywołanie `px.bar()` zawiera teraz opcjonalne argumenty `title` i `labels`.

Po wprowadzeniu zmian i uruchomieniu kodu powinieneś zobaczyć odpowiedni tytuł i etykiety dla osi, jak pokazałem na rysunku 15.13.



Rysunek 15.13. Wykres wyników symulacji rzucania pojedynczą kością typu D6 tysiąc razy

## Rzut dwiema kościemi

Rzut dwiema kościemi oznacza otrzymanie w wyniku większych liczb oraz zupełnie inny rozkład wyników. Zmodyfikujemy więc kod programu i utworzymy dwie kości typu D6, aby symulować rzut parą kości do gry. W trakcie każdego rzutu parą kości dodajemy dwie liczby (po jednej dla każdej kości), natomiast na liście `results` przechowujemy sumę wyrzuconych liczb. Kopię programu `die_visual.py` zapisz pod nazwą `dice_visual.py`, a następnie wprowadź przedstawione poniżej zmiany.

### Plik dice\_visual.py:

---

```
import plotly.express as px

from die import Die

# Utworzenie dwóch kości do gry typu D6.
die_1 = Die()
die_2 = Die()

# Wykonanie pewnej liczby rzutów i umieszczenie wyników na liście.
results = []
for roll_num in range(1000):
    result = die_1.roll() + die_2.roll() ❶
    results.append(result)

# Analiza wyników.
frequencies = []
max_result = die_1.num_sides + die_2.num_sides ❷
poss_results = range(2, max_result+1) ❸
for value in poss_results:
    frequency = results.count(value)
    frequencies.append(frequency)

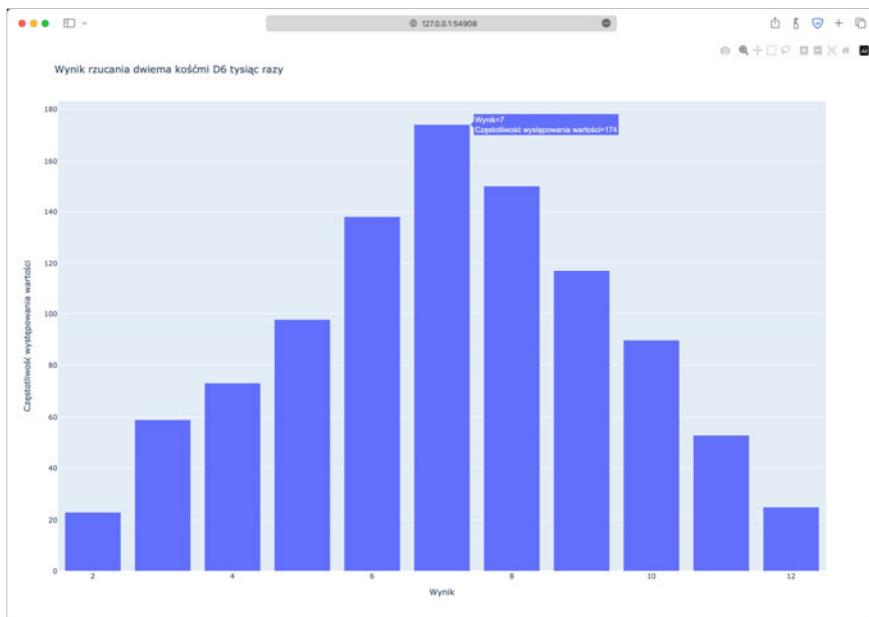
# Wizualizacja wyników.
title = "Wynik rzucania dwiema kości mi D6 tysiąc razy"
labels = {'x': 'Wynik', 'y': 'Częstotliwość występowania wartości'}
fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)
fig.show()
```

---

Po utworzeniu dwóch egzemplarzy klasy `Die` rzucamy kośćmi i obliczamy sumę otrzymanych liczb (patrz wiersz ❶). Najmniejszy możliwy wynik (2) to suma dwóch najmniejszych wyników, jakie możemy otrzymać w jednym rzucie obiema kości mi. Największy możliwy do otrzymania wynik (12) to suma dwóch największych liczb, jakie możemy otrzymać w jednym rzucie obiema kości mi. Tę wartość przechowujemy w atrybutie `max_result` (patrz wiersz ❷). Dzięki zmiennej `max_result` kod odpowiedzialny za wygenerowanie `poss_results` jest znacznie łatwiejszy w odczytcie (patrz wiersz ❸). Wprawdzie można użyć wywołania `range(2, 13)`, ale sprawdzi się ono jedynie dla dwóch kości typu D6. Podczas modelowania rzeczywistych sytuacji najlepiej tworzyć kod ułatwiający modelowanie wielu różnych ich wariancji. Powyższy kod pozwala na symulację rzutu parą kości o dowolnej liczbie ścianek.

Po uruchomieniu kodu powinieneś zobaczyć nowy wykres, podobny do pokazanego na rysunku 15.14.

Ten wykres pokazuje przybliżone wyniki, jakie prawdopodobnie otrzymasz, rzucając parą kości do gry o sześciu ściankach. Jak widzisz, najmniej prawdopodobne jest wyrzucenie wartości 2 i 12, natomiast najbardziej prawdopodobne jest wyrzucenie wartości 7, ponieważ można to zrobić na sześć sposobów: 1 i 6, 2 i 5, 3 i 4, 4 i 3, 5 i 2, 6 i 1.



Rysunek 15.14. Wykres wyników symulacji rzucania dwiema kości mi typu D6 tysiąc razy

## Dalsze dostosowanie wykresu do własnych potrzeb

Pozostała jeszcze jedna kwestia do rozwiązania po wygenerowaniu wykresu. Teraz mamy 11 słupków, a domyślne ustawienia układu dla osi X pozostawiają część z nich bez etykiet. Wprawdzie ustawienia domyślne sprawdzają się dobrze w większości wizualizacji, ale ten wykres będzie prezentował się lepiej, gdy wszystkie słupki będą miały etykiety.

Plotly oferuje metodę `update_layout()`, która pozwala wprowadzać wiele uaktualnień na wykresie po jego utworzeniu. Spójrz, jak można nakazać plotly, aby każdy słupek miał swoją etykietę.

Plik `dice_visual.py`:

---

```
--cięcie--
fig = px.bar(x=poss_results, y=frequencies, title=title, labels=labels)

# Dalsze dostosowanie wykresu do własnych potrzeb.
fig.update_layout(xaxis_dtick=1)

fig.show()
```

---

Metoda `update_layout()` działa na obiekcie `fig` przedstawiającym wykres. W omawianym przykładzie został użyty argument `xaxis_dtick`, który określa odległość między etykietami osi X. Wartość 1 oznacza, że każdy słupek będzie miał etykietę. Możesz to potwierdzić po uruchomieniu programu `dice_visual.py`.

## Rzut kośćmi o różnej liczbie ścianek

Utworzymy teraz kość o sześciu ściankach oraz drugą o dziesięciu ściankach, a następnie zasymulujemy wykonanie nimi 50 000 rzutów.

Plik dice\_visual\_d6d10.py:

---

```
import plotly.express as px

from die import Die

# Utworzenie kości typu D6 i D10.
die_1 = Die()
die_2 = Die(10) ①

# Wykonanie pewnej liczby rzutów i umieszczenie wyników na liście.
results = []
for roll_num in range(50_000):
    result = die_1.roll() + die_2.roll()
    results.append(result)

# Analiza wyników.
--cięcie--

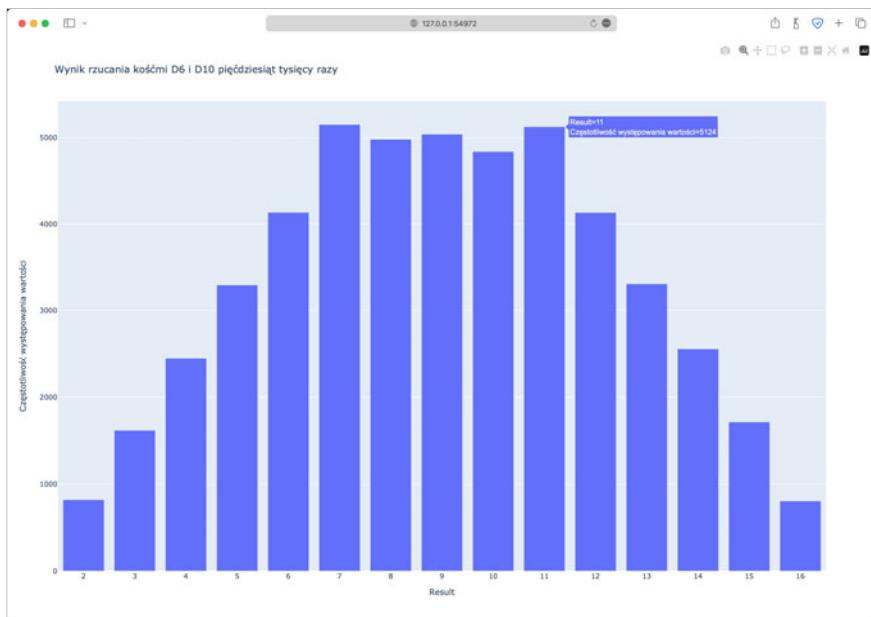
# Wizualizacja wyników.
title = "Wynik rzucania kości D6 i D10 pięćdziesiąt tysięcy razy" ②
labels = {'x': 'Result', 'y': 'Częstotliwość występowania wartości'}
--cięcie--
```

---

Aby utworzyć kość typu D10, przekazujemy argument 10 podczas definiowania drugiego egzemplarza klasy `Die` (patrz wiersz ①). Zmieniamy pierwszą pętlę, aby symulować 50 000 rzutów zamiast 1000. Dostosowujemy więc odpowiednio tytuł wykresu (patrz wiersz ②).

Na rysunku 15.15 pokazałem wykres, jaki zostanie wygenerowany. Zamiast jednego najbardziej prawdopodobnego wyniku mamy ich teraz pięć. Tak się dzieje, ponieważ wciąż jest tylko jeden sposób na wyrzucenie najmniejszej wartości (1 i 1) oraz największej (6 i 10), ale kość o mniejszej liczbie ścianek ogranicza liczbę sposobów wygenerowania środkowych wartości. Mamy sześć sposobów na wyrzucenie wartości 7, 8, 9, 10 i 11. Dlatego też to będą najczęściej otrzymywane wyniki i istnieje duże prawdopodobieństwo wyrzucenia jednego z nich.

Możliwość użycia Plotly do modelowania rzutów kościemi daje ogromną elastyczność w poznawaniu tego fenomenu. W ciągu zaledwie kilku minut możesz zasymulować ogromną liczbę rzutów wykonywanych za pomocą różnego rodzaju kości do gry.



Rysunek 15.15. Wynik 50 000 rzutów parą kości o sześciu i dziesięciu ściankach

## Zapisywanie wykresu

Gdy otrzymasz wykres, który Ci się spodoba, zawsze możesz go zapisać do pliku HTML za pomocą przeglądarki WWW. Wykres można zapisać także za pomocą polecenia w kodzie. Jeżeli chcesz zapisać wykres jako plik HTML, wywołanie `fig.show()` możesz zastąpić wywołaniem `fig.write_html()`:

```
fig.write_html('dice_visual_d6d10.html')
```

Metoda `write_html()` wymaga tylko jednego argumentu: nazwy pliku, w którym zostanie umieszczony wykres. Jeżeli podasz jedynie nazwę pliku, zostanie on zapisany w tym samym katalogu, w którym znajduje się program (plik z rozszerzeniem `.py`). Istnieje również możliwość wywołania `write_html()` z obiektem typu `Path` i zapisania pliku danych wyjściowych we wskazanym położeniu w systemie.

## ZRÓB TO SAM

**15.6. Dwie kości D8.** Przygotuj symulację pokazującą, co się stanie, jeśli 1000 razy rzucisz dwiema kości mi do gry mającymi po osiem ścianek. Spróbuj określić wygląd wizualizacji przed uruchomieniem symulacji, a później sprawdź, czy intuicja Cię nie zawiodła. Zwiększą liczbę rzutów aż do chwili, gdy zauważysz, że system zaczyna mieć trudności z wykonywaniem programu.

**15.7. Trzy kości.** Jeżeli będziesz rzucać trzema kości mi do gry typu D6, najmniejszy możliwy wynik będzie wynosić 3, natomiast największy — 18. Utwórz wizualizację pokazującą, co się stanie po rzuceniu trzema kości mi typu D6.

**15.8. Mnożenie.** Kiedy rzucasz dwiema kości mi do gry, z reguły dodajesz dwie liczby, aby otrzymać wynik. Utwórz wizualizację pokazującą, co się stanie, jeśli zamiast dodać obie liczby, pomnożysz je.

**15.9. Listy składane w kodzie.** Dla zachowania przejrzystości kodu źródłowego programy przedstawione w tym podrozdziale wykorzystywały długie pętle for. Jeżeli potrafisz używać list składanych, spróbuj zmodyfikować jedną lub obie pętle w tych programach.

**15.10. Praktyka z obiema bibliotekami.** Spróbuj użyć biblioteki matplotlib do utworzenia wizualizacji rzutów kości mi do gry oraz plotly do przygotowania wizualizacji błądzenia losowego. (Aby wykonać to ćwiczenie, musisz zajrzeć do dokumentacji obu pakietów).

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak wygenerować zbiory danych oraz tworzyć na ich podstawie wizualizacje. Zobaczyłeś, jak wygenerować proste wykresy za pomocą biblioteki matplotlib oraz jak wykorzystać wykres punktowy do przygotowania wizualizacji błądzenia losowego. Nauczyłeś się tworzyć histogram z użyciem Plotly, a także wykorzystywać ten histogram do analizy wyników rzutów kości mi o różnej liczbie ścianek.

Generowanie własnych zbiorów danych za pomocą kodu jest interesującą i potężną możliwością modelowania oraz poznawania wielu różnorodnych, rzeczywistych sytuacji. Gdy w następnych rozdziałach będziesz kontynuować pracę z projektami dotyczącymi wizualizacji danych, zwracaj uwagę na sytuacje, które można modelować z użyciem kodu. Spójrz na wizualizacje przedstawiane w mediach i zastanów się, czy potrafisz ustalić sposób ich utworzenia. Czy zostały wygenerowane za pomocą metod podobnych do metod omówionych w projektach przedstawionych w tej części książki?

W rozdziale 16. dowiesz się, jak pobierać dane z zasobów internetowych. Nadal będziemy korzystać z matplotlib i plotly do analizy tych danych.

# 16

## Pobieranie danych



W TYM ROZDZIALE BĘDZIESZ POBIERAĆ ZBIORY DANYCH ZE ŹRÓDEŁ INTERNETOWYCH ORAZ TWORZYĆ DZIAŁAJĄCE WIZUALIZACJE NA PODSTAWIE TYCH DANYCH. W INTERNECIE MOŻNA ZNALEŹĆ WRĘCZ NIEWIARYGODNIE RÓŻNORODNE DANE, Z KTÓRYCH WIĘKSZOŚĆ NIE ZOSTAŁA DOKŁADNIE PRZEANALIZOWANA. MOŻLIWOŚĆ ANALIZY TCH DANYCH POZWOLI NA ODKRYCIE WZORCÓW I POŁĄCZEŃ, KTÓRYCH NIE ODKRYLI INNI.

Uzyskamy dostęp do danych przechowywanych w dwóch popularnych formatach: CSV i JSON, a następnie stworzymy wizualizacje tych danych. Moduł Pythona o nazwie csv wykorzystamy do przetwarzania danych dotyczących pogody przechowywanych w formacie CSV (*comma-separated values*, czyli wartości rozdzielone przecinkami). Na podstawie tych danych przeanalizujemy najniższe i najwyższe temperatury zaobserwowane w dwóch różnych miejscach na przestrzeni czasu. Następnie wykorzystamy matplotlib, aby wygenerować wykres oparty na pobranych danych oraz wyświetlić odchylenia temperatury w dwóch zupełnie różniących się od siebie lokalizacjach: miejscowości Sitka (Alaska) i Dolina Śmierci (Kalifornia). W dalszej części rozdziału wykorzystamy moduł json, by uzyskać dostęp do danych o ruchach tektonicznych ziemi przechowywanych w formacie GeoJSON, i użyjemy pakietu plotly do wyświetlenia mapy pokazującej miejscowości występowania trzęsień ziemi i ich siły.

Gdy zakończysz lekturę rozdziału, będziesz przygotowany do pracy z różnymi typami i formatami zbiorów danych, a także znacznie lepiej poznasz sposoby tworzenia skomplikowanych wizualizacji. Możliwość uzyskania dostępu do różnego rodzaju danych przechowywanych w internecie oraz możliwość ich wizualizowania ma zasadnicze znaczenie podczas pracy z szerokim spektrum rzeczywistych zbiorów danych.

# Format CSV

Jednym z prostych sposobów przechowywania danych w plikach tekstowych jest ich zapisywanie jako serii *wartości rozdzielonych przecinkami*. Otrzymane w ten sposób pliki są nazywane plikami CSV. Przykładowo poniżej przedstawiłem jeden wiersz danych pogodowych zapisanych w formacie CSV:

---

```
"USW00025333","SITKA AIRPORT, AK US","2021-01-01","","44","40"
```

---

To jest fragment danych pogodowych z 1 stycznia 2021 roku dla miejscowości Sitka na Alasce. Te wartości zawierają między innymi najniższą i najwyższą temperaturę w ciągu dnia oraz wiele innych danych związanych z pogodą, jaka panowała tamtego dnia. Zawartość pliku CSV jest może trudna do odczytania dla ludzi, ale programy z łatwością mogą przetwarzarć i wyodrębniać wartości z tego formatu, co niezwykle przyśpiesza proces analizy danych.

Pracę rozpoczęliśmy od małego zbioru danych pogodowych dla wspomnianej już miejscowości Sitka na Alasce. Te dane zostały sformatowane w postaci pliku CSV i dołączone do materiałów przygotowanych do tej książki ([https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/)). W katalogu, w którym będziesz umieszczać programy tworzone w tym rozdziale utwórz podkatalog *weather\_data*, a następnie skopiuj do niego plik o nazwie *sitka\_weather\_07-2021\_simple.csv*. (Kiedy pobierzesz materiały przygotowane do tej książki, będziesz miał wszystkie pliki niezbędne do wykonania bieżącego projektu).

**UWAGA** *Dane pogodowe użyte w tym projekcie zostały pobrane ze strony <https://www.ncdc.noaa.gov/cdo-web/>.*

## Przetwarzanie nagłówków pliku CSV

Pochodzący z biblioteki standardowej moduł Pythona o nazwie *csv* przetwarza wiersze w pliku CSV i pozwala na szybkie wyodrębnienie interesujących nas wartości. Rozpoczynamy od przeanalizowania pierwszego wiersza w pliku, który będzie zawierał serię nagłówków dla danych. Te nagłówki wskazują rodzaje informacji przechowywanych przez dane.

*Plik sitka\_highs.py:*

---

```
from pathlib import Path
import csv

path = Path('weather_data/sitka_weather_07-2021_simple.csv') ❶
lines = path.read_text().splitlines()

reader = csv.reader(lines) ❷
header_row = next(reader) ❸
print(header_row)
```

---

Na początku importujemy obiekt `Path` i moduł `csv`. Następnie budujemy obiekt `Path`, sprawdzający katalog `weather_data` i prowadzący do konkretnego pliku z danymi pogody, który będzie używany w kodzie (patrz wiersz ①). Odczytujemy plik i za pomocą metody `splitlines()` otrzymujemy listę wszystkich wierszy w pliku, którą przypisujemy zmiennej `lines`. Nastepnym krokiem jest utworzenie obiektu `reader` (patrz wiersz ②). Obiekt ten będzie mógł zostać użyty do przetwarzania poszczególnych wierszy w pliku. W celu utworzenia obiektu wywołujemy funkcję `csv.reader()` i przekazujemy jej listę wierszy pobranych z pliku CSV.

W przypadku danego obiektu `reader` funkcja `next()` zwraca następny wiersz z pliku, począwszy od pierwszego. W powyższym fragmencie kodu wywołujemy `next()` tylko jednokrotnie, więc otrzymujemy jedynie pierwszy wiersz tekstu, który zawiera nagłówki pliku (patrz wiersz ③). Zwrócone dane przechowujemy w zmiennej `header_row`. Jak możesz zobaczyć, zmienna `header_row` zawiera nagłówki związane z pogodą, które informują nas o przeznaczeniu poszczególnych wartości przechowywanych w każdym wierszu danych:

---

```
['STATION', 'NAME', 'DATE', 'TAVG', 'TMAX', 'TMIN']
```

---

Obiekt `reader` przetwarza pierwszy wiersz wartości rozdzielonych przecinkami w podanym pliku i każdą z nich przechowuje jako element listy. Nagłówek `STATION` przedstawia kod stacji pogodowej, która zarejestrowała te dane. Położenie tego nagłówka informuje nas, że pierwszą wartością w każdym wierszu danych będzie kod stacji pogodowej. Nagłówek `NAME` wskazuje, że drugą wartością w każdym wierszu jest nazwa stacji pogodowej, która zarejestrowała te dane. Pozostała część nagłówka wymienia rodzaje informacji zamieszczonych w poszczególnych odczytach. Najbardziej interesujące nas dane to data zarejestrowania informacji (`DATE`), temperatura maksymalna (`TMAX`) i temperatura minimalna (`TMIN`). Jest to prosty zbiór danych zawierający jedynie informacje związane z temperaturą. Gdy samodzielnie pobierasz dane pogodowe, możesz zdecydować się na dołączenie także innych informacji, np. związanych z szybkością i kierunkiem wiatru oraz opadami.

## Wyświetlanie nagłówków i ich położenia

Aby łatwiej zrozumieć dane nagłówków pliku, każdy nagłówek wyświetlimy wraz z informacją o jego położeniu na liście.

Plik `sitka_highs.py`:

---

```
--cięcie--
reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
    print(index, column_header)
```

---

Podeczas iteracji listy funkcja `enumerate()` zwraca indeksy poszczególnych elementów, a także wartości. (Zwróć uwagę na usunięcie wiersza `print(header_row)` i zastąpienie go znacznie bardziej rozbudowaną wersją).

Poniżej przedstawiłem otrzymane dane wyjściowe, które zawierają indeksy poszczególnych nagłówków:

---

```
0 STATION
1 NAME
2 DATE
3 TAVG
4 TMAX
5 TMIN
```

---

Jak możesz zobaczyć, data oraz najwyższa temperatura danego dnia są przechowywane w kolumnach 2 i 4. Aby przeanalizować te dane, przetworzymy wszystkie wiersze danych w pliku `sitka_weather_07-2021_simple.csv` oraz wyodrębnimy wartości z indeksów 2 i 4.

## Wyodrębnienie i odczytanie danych

Skoro już wiemy, które kolumny danych są nam niezbędne, możemy przystąpić do odczytu tych danych. Przede wszystkim odczytujemy najwyższą temperaturę dla każdego dnia.

*Plik sitka\_highs.py:*

---

```
--cięcie--
reader = csv.reader(lines)
header_row = next(reader)

# Pobranie temperatur maksymalnych z pliku.
highs = [] ①
for row in reader: ②
    high = int(row[4]) ③
    highs.append(high)

print(highs)
```

---

Tworzymy pustą listę o nazwie `highs` (patrz wiersz ①), a następnie przeprowadzamy iterację przez pozostałe wiersze w pliku (patrz wiersz ②). Obiekt `reader` będzie kontynuował działanie od miejsca ostatniej operacji w pliku CSV oraz automatycznie zwróci każdy wiersz znajdujący się po aktualnie zwracanym. Ponieważ wiersz nagłówka został już odczytany, działanie pętli będzie kontynuowane od wiersza drugiego, w którym zaczynają się rzeczywiste dane. W trakcie każdej iteracji pętli dane pochodzące z indeksu 4, czyli odpowiadającego kolumnie `TMAX`, są przypisywane zmiennej `high` (patrz wiersz ③). Funkcja `int()` została użyta do konwersji danych z ciągu tekstopowego, w którym są przechowywane, na

postać liczb całkowitych możliwych później do użycia. Następnie ta wartość jest dołączana do tablicy highs.

Poniżej pokazałem dane, które na obecnym etapie są przechowywane w highs:

---

```
[61, 60, 66, 60, 65, 59, 58, 58, 57, 60, 60, 60, 57, 58, 60, 61, 63, 63, 70, 64, 59, 63, 61, 58, 59, 64, 62, 70, 70, 73, 66]
```

---

W ten sposób wyodrębniliśmy najwyższe temperatury zanotowane w poszczególnych dniach i przechowujemy je elegancko na liście w postaci liczb. Przystępujemy teraz do utworzenia wizualizacji dla tych danych.

## Wyświetlenie danych na wykresie temperatury

Aby przygotować wizualizację zebranych danych temperatury, przede wszystkim musimy za pomocą biblioteki matplotlib utworzyć prosty wykres najwyższych temperatur dla poszczególnych dni, tak jak pokazałem w poniższym fragmencie kodu.

*Plik sitka\_highs.py:*

---

```
from pathlib import Path
import csv

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
lines = path.read_text().splitlines()
--cięcie--

# Dane wykresu.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(highs, color='red') ❶

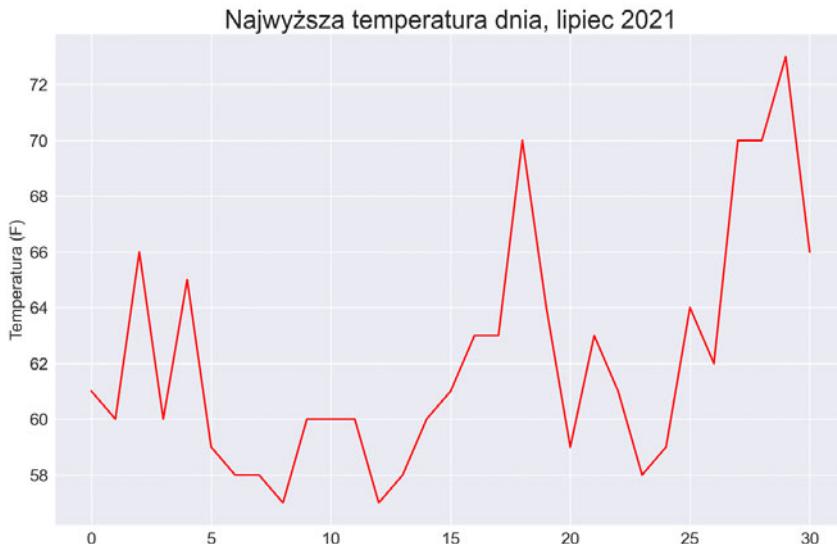
# Formatowanie wykresu.
ax.set_title("Najwyższa temperatura dnia, lipiec 2021", fontsize=24) ❷
ax.set_xlabel('', fontsize=16) ❸
ax.set_ylabel("Temperatura (F)", fontsize=16)
ax.tick_params(labelsize=16)

plt.show()
```

---

W wierszu ❶ dane w postaci najwyższych temperatur dla poszczególnych dni przekazujemy metodzie plot(). Ponadto przekazujemy parametr color='red', aby punkty zostały wyświetlane w kolorze czerwonym. (Najwyższe temperatury wyświetlimy na czerwono, natomiast najniższe na niebiesko). Następnie podajemy kilka innych szczegółów formatowania, między innymi tytuł wykresu, wielkość czcionki oraz etykiety (patrz wiersz ❷); te elementy powinieneś już znać z poprzedniego rozdziału. Ponieważ musimy jeszcze dodać daty, nie będziemy

na razie nadawać etykiety osi X, ale w wywołaniu `ax.set_xlabel()` zmienimy wielkość czcionki, aby etykiety domyślne były znacznie czytelniejsze (patrz wiersz ❸). Na rysunku 16.1 pokazałem wykres otrzymany w wyniku działania omawianego kodu. Jest to prosty wykres liniowy pokazujący najwyższe temperatury zanotowane w miejscowości Sitka (Alaska) w lipcu 2021 roku.



Rysunek 16.1. Wykres liniowy pokazujący najwyższe temperatury w poszczególnych dniach lipca 2021 roku w miejscowości Sitka

## Moduł `datetime`

Dodamy teraz daty do wykresu, aby tym samym stał się bardziej użyteczny. Pierwsza data w pliku danych pogodowych znajduje się w drugim wierszu:

```
"USW00025333","SITKA AIRPORT, AK US","2021-07-01","","61","53"
```

Data zostanie odczytana jako ciąg tekstowy, więc konieczne jest znalezienie sposobu konwersji ciągu tekstowego "2021-07-01" na obiekt przedstawiający tę datę. Obiekt przedstawiający datę „1 lipca 2021 roku” możemy przygotować za pomocą metody `strptime()` z modułu `datetime`. Poniżej pokazałem sposób działania `strptime()` w sesji Pythona w powłoce:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2021-07-01', '%Y-%m-%d')
>>> print(first_date)
2021-07-01 00:00:00
```

Najpierw importujemy klasę `datetime` z modułu `datetime`. Następnie wywołujemy metodę `strptime()` wraz z ciągiem tekstowym zawierającym interesującą nas datę podaną jako pierwszy argument. Drugi argument wskazuje Pythonowi sposób formatowania daty. W omawianym przykładzie ciąg tekstowy formatowania '`%Y-`' nakazuje Pythonowi zinterpretować fragment ciągu tekstu przed pierwszym łącznikiem jako rok wyrażony za pomocą czterech cyfr. Z kolei '`%m-`' nakazuje Pythonowi zinterpretować fragment ciągu tekstu przed drugim łącznikiem jako numer miesiąca. Natomiast '`%d`' nakazuje Pythonowi zinterpretować pozostały fragment ciągu tekstu jako dzień miesiąca wyrażony za pomocą cyfr od 01 do 31.

Metoda `strptime()` może pobierać wiele różnych argumentów w celu ustalenia sposobu interpretacji daty. W tabeli 16.1 wymieniłem niektóre z tych argumentów.

*Tabela 16.1. Oferowane przez moduł `datetime` argumenty przeznaczone do formatowania daty i godziny*

Argument	Opis
<code>%A</code>	Nazwa dni tygodnia, na przykład poniedziałek.
<code>%B</code>	Nazwa miesiąca, na przykład styczeń.
<code>%m</code>	Miesiąc wyrażony cyframi (od 01 do 12).
<code>%d</code>	Dzień miesiąca wyrażony cyframi (od 01 do 31).
<code>%Y</code>	Rok w postaci czterocyfrowej liczby, na przykład 2019.
<code>%y</code>	Rok w postaci dwucyfrowej liczby, na przykład 19.
<code>%H</code>	Godzina w formacie 24-godzinnym (od 00 do 23).
<code>%I</code>	Godzina w formacie 12-godzinnym (od 01 do 12).
<code>%p</code>	Wyświetla AM lub PM.
<code>%M</code>	Minuty (od 00 do 59).
<code>%S</code>	Sekundy (od 00 do 59).

## Wyświetlanie daty

Znając sposoby przetwarzania dat zapisanych w pliku CSV, możemy poprawić wykres danych temperatur przez wyodrębnienie dat i ich wyświetlenie na osi X, jak pokazałem w poniższym fragmencie kodu.

*Plik sitka\_highs.py:*

```
from pathlib import Path
import csv
from datetime import datetime

import matplotlib.pyplot as plt

path = Path('weather_data/sitka_weather_07-2021_simple.csv')
```

```

lines = path.read_text().splitlines()
reader = csv.reader(lines)
header_row = next(reader)

# Pobranie dat i najwyższych temperatur z pliku.
dates, highs = [], [] ❶
for row in reader:
    current_date = datetime.strptime(row[2], "%Y-%m-%d") ❷
    high = int(row[4])
    dates.append(current_date)
    highs.append(high)

# Wygenerowanie wykresu najwyższych temperatur.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(dates, highs, color='red') ❸

# Formatowanie wykresu.
ax.set_title("Najwyższa temperatura dnia, lipiec 2021", fontsize=24)
ax.set_xlabel('', fontsize=16)
fig.autofmt_xdate() ❹
ax.set_ylabel("Temperatura (F)", fontsize=16)
ax.tick_params(labelsize=16)

plt.show()

```

---

W wierszu ❶ tworzymy puste listy przeznaczone do przechowywania dat oraz najwyższych temperatur odnotowanych w poszczególnych dniach (te dane są pobierane z pliku). Następnie dane zawierające te informacje (row[2]) konwertujemy na obiekt `datetime` (patrz wiersz ❷) i dodajemy do `dates`. W wierszu ❸ daty i wartości najwyższych temperatur przekazujemy do funkcji `plot()`. Wywołanie `fig.autofmt_xdate()` wyświetla etykiety ukośnie, aby uniknąć ich nakładania się na siebie (patrz wiersz ❹). Na rysunku 16.2 pokazałem poprawiony wykres.

## Wyświetlenie dłuższego przedziału czasu

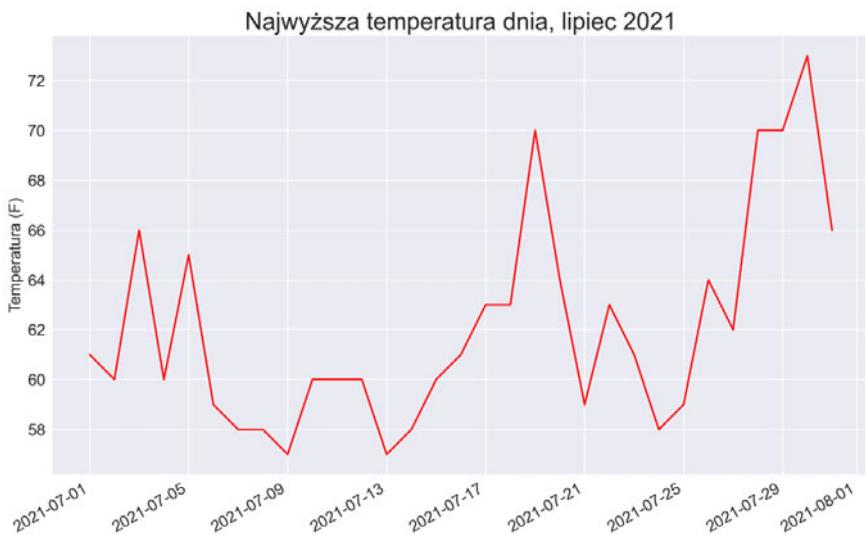
Mając skonfigurowany wykres, dodamy do niego większą ilość danych i tym samym otrzymamy znacznie dokładniejsze informacje o pogodzie w miejscowości Sitka. Do podkatalogu `data` w katalogu zawierającego programy tworzone w tym rozdziale skopiuj plik `sitka_weather_2021_simple.csv`, w którym znajdują się dane pogodowe dla miejscowości Sitka dotyczące całego 2021 roku.

Teraz możemy wygenerować wykres pokazujący najwyższe temperatury w całym roku.

*Plik sitka\_highs.py:*

---

```
--cięcie--
path = Path('weather_data/sitka_weather_2021_simple.csv') lines =
path.read_text().splitlines()
```



Rysunek 16.2. Wykres jest teraz znacznie czytelniejszy, zawiera daty podane na osi X

```
--cięcie--
# Formatowanie wykresu.
ax.set_title("Najwyższa temperatura dnia - 2012", fontsize=24)
ax.set_xlabel('', fontsize=16)
--cięcie--
```

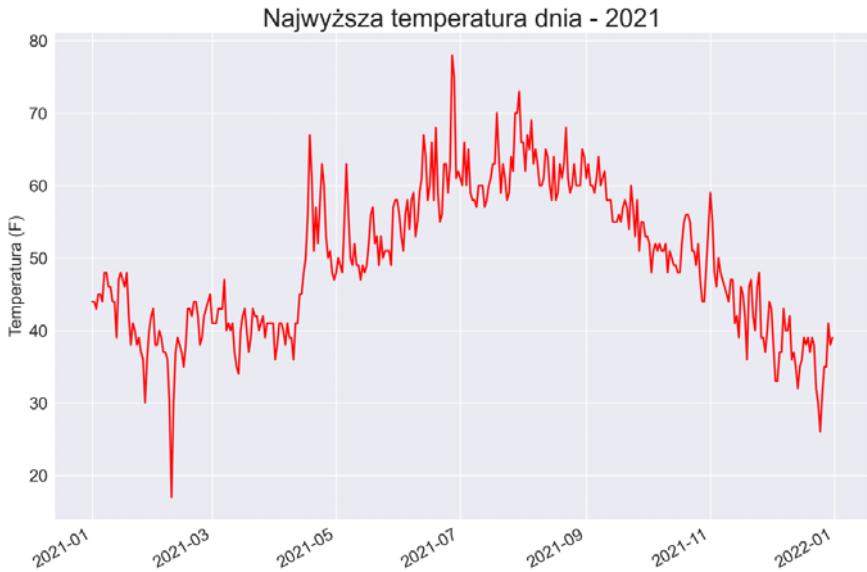
Modyfikujemy nazwę pliku, ponieważ teraz będziemy używać pliku `sitka_weather_2021_simple.csv` zawierającego dane pogodowe dla całego 2021 roku. Uaktualniamy także tytuł wykresu, aby odzwierciedlał tę zmianę prezentowanej zawartości. Wygenerowany wykres pokazalem na rysunku 16.3.

## Wyświetlenie drugiej serii danych

Wprawdzie zmodyfikowany wykres pokazany na rysunku 16.3 zawiera wiele przydatnych danych, ale jego użyteczność możemy jeszcze bardziej zwiększyć przez umieszczenie informacji o najniższej temperaturze w poszczególnych dniach. Najniższe temperatury trzeba wyodrębnić z pliku danych, a następnie dodać do wykresu, jak pokazałem w poniższym fragmencie kodu.

Plik `sitka_highs_lows.py`:

```
--cięcie--
reader = csv.reader(lines)
header_row = next(reader)
```



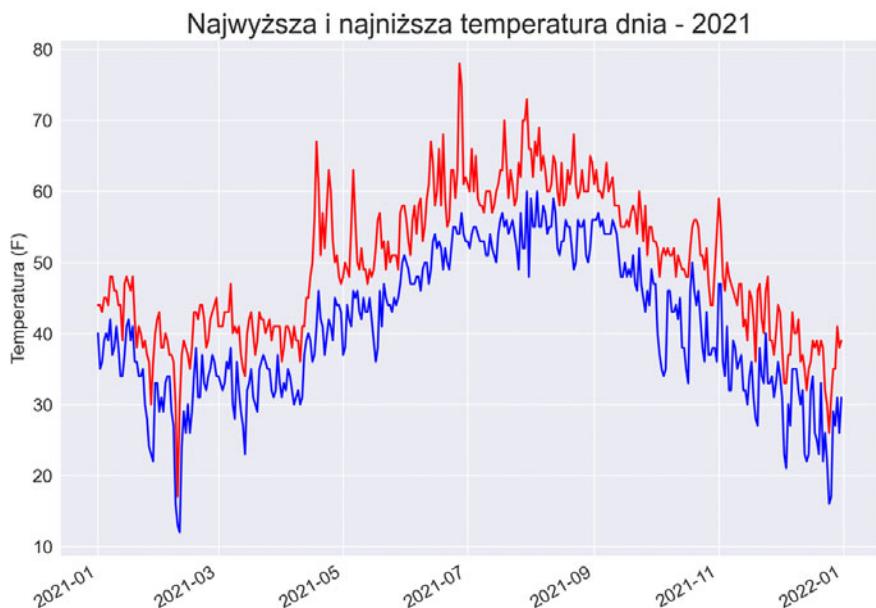
Rysunek 16.3. Dane pogodowe zebrane w 2021 roku

```
# Pobranie dat oraz najwyższych i najniższych temperatur z pliku.
dates, highs, lows = [], [], [] ❶
for row in reader:
    current_date = datetime.strptime(row[2], "%Y-%m-%d")
    high = int(row[4])
    low = int(row[5]) ❷
    dates.append(current_date)
    highs.append(high)
    lows.append(low)

# Wygenerowanie wykresu najwyższych i najniższych temperatur.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(dates, highs, color='red')
ax.plot(dates, lows, color='blue') ❸

# Formatowanie wykresu.
ax.set_title("Najwyższa i najniższa temperatura dnia - 2021", fontsize=24) ❹
--cięcie--
```

W wierszu ❶ dodajemy pustą listę `lows` przeznaczoną do przechowywania najniższych temperatur, a następnie z położenia `row[5]` w każdym wierszu wyodrębniamy najniższą temperaturę dla każdej daty (patrz wiersz ❷). W wierszu ❸ dodajemy wywołanie `plot()` dla najniższych temperatur i definiujemy dla nich kolor niebieski. Na koniec aktualniamy tytuł (patrz wiersz ❹). Ostateczną postać wykresu pokazałem na rysunku 16.4.



Rysunek 16.4. Dwie serie danych na tym samym wykresie

## Nakładanie cienia na wykresie

Kiedy mamy na wykresie dwie serie danych, możemy przeanalizować zakres temperatur dla poszczególnych dni. Dokonujemy więc pracy nad wykresem przez nałożenie cienia pokazującego zakres między najniższą i najwyższą temperaturą w danym dniu. W tym celu użyjemy metody `fill_between()` pobierającej serie wartości osi X oraz dwie serie wartości osi Y i wypełniającej obszar między tymi dwiema seriami wartości dla osi Y.

Plik `sitka_highs_lows.py`:

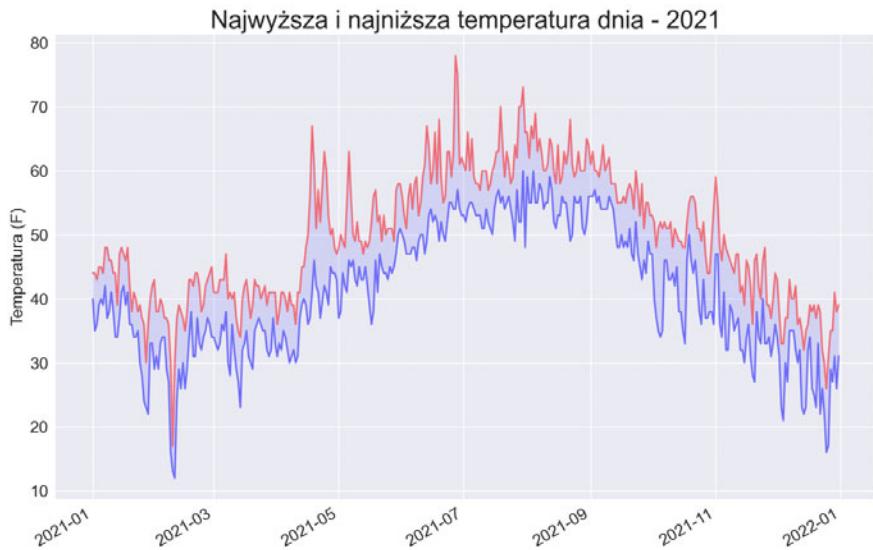
---

```
--cięcie--
# Wygenerowanie wykresu najniższych i najwyższych temperatur.
plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.plot(dates, highs, color='red', alpha=0.5) ❶
ax.plot(dates, lows, color='blue', alpha=0.5)
ax.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1) ❷
--cięcie--
```

---

Argument `alpha` w wierszu ❶ określa przezroczystość koloru. Wartość `alpha` wynosząca 0 oznacza całkowitą przezroczystość, natomiast 1 (wartość domyślna) całkowite pokrycie kolorem. Przypisujemy argumentowi `alpha` wartość 0.5 i sprawiamy tym samym, że czerwone i niebieskie linie wykresu są nieco delikatniejsze.

W wierszu ❷ przekazujemy wywołaniu `fill_between()` listę `dates` jako serię wartości dla osi X oraz listy `highs` i `lows` jako dwie serie wartości dla osi Y. Argument `facecolor` określa kolor obszaru, na którym będzie znajdował się cień. Definiujemy dla niego niską wartość `alpha` wynoszącą 0.1, aby wypełniony region łączył dwie serie danych bez niepotrzebnego odciągania uwagi od prezentowanych informacji. Na rysunku 16.5 pokazałem wykres wraz z pocieniowanym obszarem między najniższą i najwyższą temperaturą danego dnia.



Rysunek 16.5. Obszar między dwoma zbiorami danych jest pocieniowany

Cieniowanie powoduje, że obszar znajdujący się między dwiema seriami danych staje się natychmiast widoczny.

## Sprawdzenie pod kątem błędów

Opracowana tutaj wersja programu `sitka_highs_lows.py` pozwala na użycie danych pogodowych dla dowolnej miejscowości. Jednak niektóre stacje pogody czasami działają nieprawidłowo i nie zbierają poprawnie pewnych, lub nawet wszystkich oczekiwanych danych. Brakujące dane mogą doprowadzić do awarii programu, jeśli tego rodzaju sytuacja nie zostanie prawidłowo obsłużona.

Zobaczmy na przykład, co się stanie, jeśli spróbujemy wygenerować wykres temperatury dla Doliny Śmierci (Kalifornia). Do podkatalogu `data` w katalogu zawierającym programy utworzone w tym rozdziale skopiuj plik `death_valley_2021_simple.csv`.

Zacznij od wykonania przedstawionego tutaj kodu, aby poznać nagłówki znajdujące się w pliku danych.

### *Plik death\_valley\_highs\_lows.py:*

---

```
from pathlib import Path
import csv

path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()

reader = csv.reader(lines)
header_row = next(reader)

for index, column_header in enumerate(header_row):
    print(index, column_header)
```

---

Oto otrzymane dane wyjściowe, które zawierają indeksy poszczególnych nagłówków:

---

```
0 STATION
1 NAME
2 DATE
3 TMAX
4 TMIN
5 TOBS
```

---

Jak widzisz, informacje o dacie rejestracji pomiaru znajdują się w tym samym położeniu (o indeksie 2) co w przypadku danych pogodowych dla miejscowości Sitka. Natomiast wartości temperatury najwyższej i najmniejszej są w indeksach 3 i 4, więc konieczne będzie wprowadzenie zmian w kodzie odzwierciedlających rzeczywiste położenie danych. Zamiast dostarczać średnią temperaturę w danym dniu ta stacja pogodowa umieszcza w danych nagłówek TOBS zawierający godzinę przeprowadzenia odczytu.

Zmodyfikuj kod z pliku *sitka\_highs\_lows.py* w celu wygenerowania wykresu dla Doliny Śmierci, używając odpowiednich numerów indeksów, i zobacz, co się stanie po uruchomieniu programu.

### *Plik death\_valley\_highs\_lows.py:*

---

```
--cięcie--
path = Path('weather_data/death_valley_2021_simple.csv')
lines = path.read_text().splitlines()
--cięcie--
# Pobranie dat oraz najwyższych i najniższych temperatur z pliku.
dates, highs, lows = [], [], []
for row in reader:
    current_date = datetime.strptime(row[2], "%Y-%m-%d")
    high = int(row[3])
    low = int(row[4])
    dates.append(current_date)
--cięcie--
```

---

Program zawiera uaktualnione indeksy odpowiadające położeniom nagłówków TMAX i TMIN w pliku danych pogodowych dla Doliny Śmierci.

Jednak po uruchomieniu programu otrzymasz błąd:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 17, in <module>
    high = int(row[3])
ValueError: invalid literal for int() with base 10: '' ①
```

Ten stos wywołań informuje nas, że Python nie mógł przetworzyć najwyższej temperatury dla jednego z dni, ponieważ nie potrafi skonwertować pustego ciągu tekstuowego (' ') na postać liczby całkowitej (patrz wiersz ①). Zamiast analizować dane i próbować ustalić, których informacji brakuje, lepiej bezpośrednio zająć się obsługą brakujących danych.

Rozwiązaniem tego rodzaju problemu jest przygotowanie kodu, który podczas odczytywania wartości z pliku CSV będzie je sprawdzał pod kątem błędów. W ten sposób będzie można obsługiwać wyjątki, które mogą zostać zgłoszone w trakcie przetwarzania zbiorów danych. Poniżej przedstawiłem odpowiedni fragment kodu.

#### Plik death\_valley\_highs\_lows.py:

```
--cięcie--
for row in reader:
    current_date = datetime.strptime(row[2], "%Y-%m-%d")
    try: ①
        high = int(row[3])
        low = int(row[4])
    except ValueError:
        print(f"Brak danych dla {current_date}.") ②
    else: ③
        dates.append(current_date)
        highs.append(high)
        lows.append(low)

# Wygenerowanie wykresu najniższych i najwyższych temperatur.
--cięcie--

# Formatowanie wykresu.
title = "Najwyższa i najniższa temperatura dnia - 2021\nDolina Śmierci, Kalifornia" ④
plt.title(title, fontsize=20)
plt.xlabel(' ', fontsize=16)
--cięcie--
```

Analizując każdy wiersz danych, próbujemy wyodrębnić datę oraz najniższą i najwyższą temperaturę danego dnia (patrz wiersz ①). Jeżeli będzie brakowało jakichkolwiek danych, Python zgłosi wyjątek ValueError, który obsłużymy przez wyświetlenie komunikatu błędu, zawierającego informację o brakujących danych

(patrz wiersz ②). Po wyświetleniu komunikatu błędu pętla będzie kontynuowała działanie i przetworzy następny wiersz danych. Jeżeli wszystkie dane dla analizowanego dnia zostaną pobrane bezbłędnie, zostanie wykonany blok `else` i dane zostaną dołączone do odpowiednich list (patrz wiersz ③). Ponieważ wyświetlamy teraz dane dla innej lokalizacji, uaktualniamy tytuł wykresu i umieszczaemy w nim informację o nowym miejscu (patrz wiersz ④).

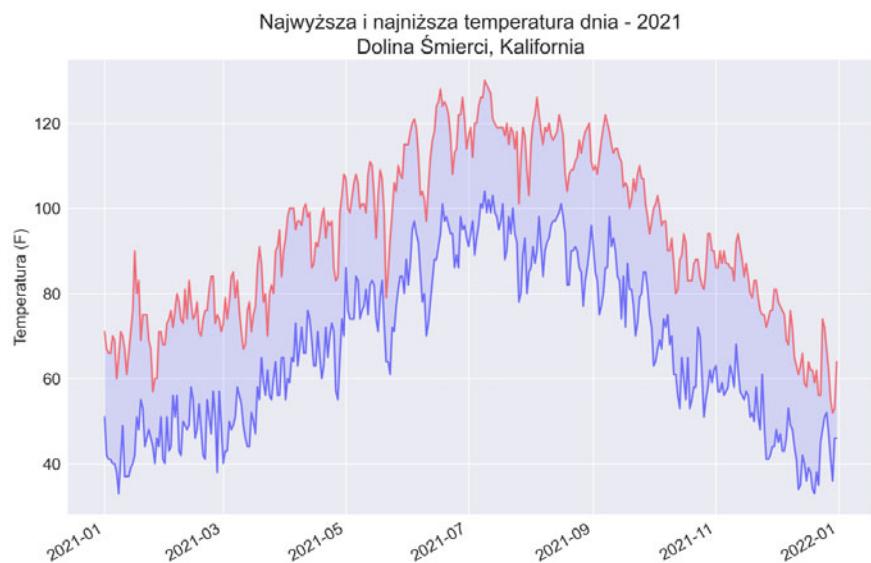
Jeżeli uruchomisz program `death_valley_highs_lows.py`, otrzymasz informacje o brakujących danych tylko dla jednego dnia:

---

Brak danych dla 2021-05-04 00:00:00.

---

Ponieważ błąd zostanie odpowiednio obsłużony, kod będzie mógł wygenerować wykres z pominięciem brakujących danych. Wygenerowany wykres pokazałem na rysunku 16.6.



Rysunek 16.6. Najniższe i najwyższe temperatury w poszczególnych dniach 2021 roku w Dolinie Śmierci

Jeżeli porównamy powyższy wykres z wygenerowanym wcześniej wykresem dla miejscowości Sitka, zauważmy, że Dolina Śmierci jest znacznie cieplejsza niż południowo-wschodnia Alaska, co jest oczywiste. Na wykresie widać również, że dobowa amplituda temperatury jest znacznie większa na pustyni. Wielkość zacienionych obszarów na wykresie wyraźnie to pokazuje.

Wiele zbiorów danych, z którymi przyjdzie Ci pracować, będzie miało braki w danych, niepoprawnie sformatowane, lub wręcz nieprawidłowe dane. Narzędzia

poznanie w części pierwszej książki wykorzystaj do rozwiązywania tego rodzaju problemów. W tym rozdziale skorzystaliśmy z konstrukcji `try-except-else` do obsługi brakujących danych. Czasami można użyć polecenia `continue` i pominąć pewne dane, ewentualnie za pomocą wywołania `remove()` lub `del` pozbyć się danych już po ich wyodrębnieniu. Możesz wykorzystać dowolne sprawdzające się podejście, o ile pozwoli Ci otrzymać sensowną i prawidłową wizualizację.

## Samodzielne pobieranie danych

Jeżeli chcesz pobrać inne dane pogodowe, skorzystaj z następującej procedury:

1. Odwiedź witrynę NOAA Climate Data Online znajdująca się pod adresem <https://www.ncdc.noaa.gov/cdo-web/>. W sekcji *Discover Data By* kliknij *Search Tool*. Następnie w polu *Select a Dataset* wybierz *Daily Summaries*.
2. W sekcji *Search For* określ zakres dat i wybierz *ZIP Codes*.  
Podaj interesujący Cię kod pocztowy i kliknij przycisk *Search*.
3. Na kolejnej stronie zobaczysz mapę i informacje o obszarze, na którym się koncentrujesz. Poniżej nazwy lokalizacji kliknij *View Full Details* lub najpierw kliknij mapę, a później *Full Details*.
4. Przewini stronę do dołu i kliknij *Station List*, aby poznać stacje pogodowe dostępne na danym obszarze. Po wybraniu jednej ze stacji kliknij *Add to Cart*. Dane są dostępne bezpłatnie, pomimo że w witrynie jest używana ikona koszyka na zakupy. Kliknij koszyk na zakupy w prawym górnym rogu.
5. W sekcji *Select the Output* wybierz *Custom GHCN-Daily CSV*. Upewnij się, czy został podany prawidłowy zakres dat, i kliknij przycisk *Continue*.
6. Na następnej stronie możesz wybrać rodzaj interesujących Cię danych. Możesz pobrać tylko jeden rodzaj danych, np. tylko dane dotyczące temperatury powietrza, lub wszystkie zebrane przez daną stację.  
Dokonaj wyboru i kliknij przycisk *Continue*.
7. Na ostatniej stronie zostanie wyświetlone podsumowanie zamówienia.  
Podaj adres e-mail i kliknij przycisk *Submit Order*. Otrzymasz powiadomienie o złożeniu zamówienia, a po kilku minutach kolejną wiadomość e-mail – z łączem umożliwiającym pobranie żądanych danych.

Dane, które pobierzesz, będą miały tę samą strukturę co dane wykorzystane w tym podrozdziale. Nagłówki mogą się różnić od tych, które widziałeś w omawianych tutaj programach. Jeżeli jednak będziesz wykonywać kroki przedstawione w tym rozdziale, powinieneś mieć możliwość wygenerowania wizualizacji interesujących Cię danych.

## ZRÓB TO SAM

**16.1. Opady deszczu w Sitka.** Miejscowość Sitka leży w strefie klimatów umiarkowanych, więc dość często pada tam deszcz. W pliku *sitka\_weather\_2021\_simple.csv* znajduje się nagłówek o nazwie PRCP, który przedstawia wielkość dziennych opadów. Utwórz wizualizację koncentrującą się na danych w tej kolumnie. To ćwiczenie możesz wykonać również dla Doliny Śmierci, jeśli chcesz poznać wielkość opadów deszczu na pustyni.

**16.2. Porównanie miejscowości Sitka i Doliny Śmierci.** Skale temperatur na wykresach utworzonych dla miejscowości Sitka i Doliny Śmierci mają różne zakresy liczbowe. Aby dokładniej porównać rozkład temperatur w Sitka i Dolinie Śmierci, musisz zastosować taką samą skalę dla osi Y. Zmień ustawienia dla osi Y na jednym lub na obu wykresach pokazanych na rysunkach 16.5 i 16.6, a będziesz mógł przeprowadzić bezpośrednie porównanie rozkładu temperatur dla miejscowości Sitka i Dolinie Śmierci (lub każdych innych dwóch lokalizacji).

**16.3. San Francisco.** Czy temperatury w San Francisco bardziej przypominają te zaobserwowane w miejscowości Sitka, czy raczej w Dolinie Śmierci? Pobierz dane dla San Francisco, a następnie wygeneruj wykres najniższych i najwyższych temperatur dla poszczególnych dni roku i dokonaj porównania.

**16.4. Indeksy automatyczne.** W podrozdziale na stałe zostały podane indeksy odpowiadające kolumnom TMIN i TMAX. Wiersz nagłówka wykorzystaj do ustalenia numerów indeksów dla tych wartości, aby program mógł działać z danymi dotyczącymi zarówno miejscowości Sitka, jak i Doliny Śmierci. Nazwę stacji wykorzystaj do automatycznego wygenerowania tytułu dla wykresu.

**16.5. Eksperymentuj.** Wygeneruj kilka następnych wizualizacji, aby przeanalizować jeszcze inne interesujące Cię aspekty pogody w dowolnie wybranych lokalizacjach.

## Mapowanie globalnych zbiorów danych – format GeoJSON

W tym podrozdziale pobierzemy dane przedstawiające aktywność sejsmiczną ziemi w ubiegłym miesiącu. Następnie utworzymy mapę pokazującą miejsca wystąpień trzęsień ziemi i ich siłę. Ponieważ te dane są przechowywane w formacie GeoJSON, będziemy z nimi pracować za pomocą modułu json. Używając oferowanego przez plotly wywołania `scatter_geo()`, utworzymy wizualizację tych danych pokazującą globalne wzorce rozkładu trzęsień ziemi.

# Pobranie danych dotyczących trzęsień ziemi

Do podkatalogu `eq_data` w katalogu zawierającym programy tworzone w tym rozdziale skopiuj plik o nazwie `eq_1_day_m1.geojson`. Trzęsienia ziemi są kategoryzowane na podstawie ich siły wyrażonej w skali Richtera. W tym pliku znajdują się dane o wszystkich trzęsieniach ziemi o sile co najmniej M1, które miały miejsce w ciągu ostatnich 24 godzin od chwili, gdy pisałam te słowa. Te dane pochodzą z jednego ze zbiorów danych udostępnionych bezpłatnie przez United States Geological Survey (<https://earthquake.usgs.gov/earthquakes/feed/>).

## Analiza danych GeoJSON

Gdy otworzysz plik `eq_1_day_m1.geojson`, zobaczysz, że dane są dość mocno upakowane i trudne w odczycie.

---

```
{"type": "FeatureCollection", "metadata": {"generated": 1649052296000, ...  
{"type": "Feature", "properties": {"mag": 1.6, "place": "63 km SE of Ped...  
{"type": "Feature", "properties": {"mag": 2.2, "place": "27 km SSE of Ca...  
{"type": "Feature", "properties": {"mag": 3.7, "place": "102 km SSE of S...  
{"type": "Feature", "properties": {"mag": 2.92000008, "place": "49 km SE...  
{"type": "Feature", "properties": {"mag": 1.4, "place": "44 km NE of Sus...  
--cięcie--
```

---

Ten plik został sformatowany dla urządzeń, a nie z myślą o odczycie przez człowieka. Mimo to można dostrzec, że zawiera pewne słowniki, a także interesujące nas informacje, takie jak miejsce trzęsienia ziemi i jego siłę.

Moduł `json` oferuje różne narzędzia przeznaczone do analizowania danych JSON i pracy z nimi. Część tych narzędzi pomoże w ponownym sformatowaniu pliku, aby można było znacznie łatwiej dostrzec interesujące nas dane przed rozpoczęciem programistycznej pracy z nimi.

Zaczynamy od wczytania pliku i wyświetlenia go w formacie łatwiejszym do odczytu. Plik zawiera wiele danych, więc zamiast je wyświetlić na ekranie zostaną zapisane w nowym pliku. Następnie będzie można otworzyć nowy plik i przeglądać jego zawartość.

*Plik `eq_explore_data.py`:*

---

```
from pathlib import Path  
import json  
  
# Odczytanie danych jako ciągu tekstowego i ich konwersja na obiekt Pythona.  
path = Path('eq_data/eq_data_1_day_m1.geojson')  
contents = path.read_text()  
all_eq_data = json.loads(contents) ❶
```

```
# Utworzenie znacznie czytelniejszej wersji pliku danych.  
path = Path('eq_data/readable_eq_data.geojson') ❷  
readable_contents = json.dumps(all_eq_data, indent=4) ❸  
path.write_text(readable_contents)
```

---

Zaczynamy od zimportowania pliku danych jako ciągu tekstowego. Następnie funkcja `json.loads()` konwertuje dane na format możliwy do użycia w Pythonie ❶. Dokładnie takie samo podejście zostało zastosowane w rozdziale 10. W omawianym przykładzie cały zbiór danych został skonwertowany na pojedynczy słownik, który następnie zostaje umieszczony w zmiennej `all_eq_data`. Kolejnym krokiem jest zdefiniowanie obiektu `path` pozwalającego zapisać te same dane, ale w znacznie czytelniejszym formacie ❷. Znana z rozdziału 10. funkcja `json.dumps()` może pobierać opcjonalny argument `indent` (patrz wiersz ❸), który nakazuje jej sformatowanie danych z wykorzystaniem wcięć dopasowanych do struktury danych.

Gdy zajrzesz do katalogu `eq_data` i otworzysz plik `readable_eq_data.json`, początek danych będzie przedstawiał się następująco:

```
{  
    "type": "FeatureCollection",  
    "metadata": { ❶  
        "generated": 1649052296000,  
        "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",  
        "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",  
        "status": 200,  
        "api": "1.10.3",  
        "count": 160  
    },  
    "features": [ ❷  
--cięcie--
```

---

Część pierwsza pliku zawiera sekcję z kluczem `"metadata"` (patrz wiersz ❶). Znajdują się tutaj informacje o tym, kiedy plik został wygenerowany i gdzie w internecie można znaleźć odpowiednie dane. Ta sekcja zawiera również czytelny dla człowieka tytuł i informacje o liczbie trzęsień ziemi uwzględnionych w tym pliku. W pliku znajdują się dane o 160 trzęsieniach ziemi zarejestrowanych w ciągu 24 godzin.

Ten plik GeoJSON ma strukturę przydatną dla danych opartych na lokalizacji. Informacje są zamieszczone na liście powiązanej z kluczem `"features"` (patrz wiersz ❷). Ponieważ plik zawiera dane związane z trzęsieniami ziemi, dane znajdują się na liście, której każdy element odpowiada jednemu trzęsieniu ziemi. Wprawdzie taka struktura może wydawać się zagmatwana, ale jest bardzo przydatna. Umożliwia geologom przechowywanie w słowniku niezbędnej ilości informacji o poszczególnych trzęsieniach, a następnie umieszczenie wszystkich słówników na jednej ogromnej liście.

Spójrz na słownik przedstawiający pojedyncze trzęsienie ziemi.

## Plik readable\_eq\_data.json:

---

```
--cięcie--  
{  
    "type": "Feature",  
    "properties": { ❶  
        "mag": 1.6,  
        --cięcie--  
        "title": "M 1.6 - 27 km NNW of Susitna, Alaska" ❷  
    },  
    "geometry": { ❸  
        "type": "Point",  
        "coordinates": [  
            -150.7585, ❹  
            61.7591, ❺  
            56.3  
        ] },  
        "id": "ak0224bjuljx"  
}
```

---

Klucz "properties" zawiera wiele informacji o każdym trzęsieniu ziemi (patrz wiersz ❶). Przede wszystkim interesuje nas siła każdego trzęsienia, która jest zapisana jako wartość klucza "mag". Ponadto interesuje nas tytuł zarejestrowanego zdarzania dostarczający eleganckie podsumowanie o sile i miejscu wystąpienia danego trzęsienia ziemi (patrz wiersz ❷).

Klucz "geometry" pomaga poznać miejsce wystąpienia trzęsienia ziemi (patrz wiersz ❸). Te informacje będą potrzebne do utworzenia mapy zdarzeń. Długość (patrz wiersz ❹) i szerokość (patrz wiersz ❺) geograficzną poszczególnych trzęsień ziemi można odczytać z listy przypisanej jako wartość klucza "coordinates".

Ten plik zawiera znacznie bardziej zagnieźdzone dane niż używane w dotąd tworzonym kodzie, więc jeśli treść pliku wygląda niezrozumiale, to się tym nie przejmuj. Python zajmie się obsługą całe tej złożoności. W danym momencie będziemy pracować z tylko jednym lub dwoma poziomami zagnieżdżenia. Zaczniemy od wyodrębnienia słownika dla każdego trzęsienia ziemi odnotowanego w ciągu 24 godzin.

**UWAGA** Gdy będę mówił o lokalizacji, często podam najpierw szerokość, a dopiero później długość geograficzną. Ta konwencja prawdopodobnie wynika stąd, że ludzkość odkryła szerokość geograficzną na długo przed tym, zanim opracowano koncepcję długości geograficznej. Jednak wiele frameworków przeznaczonych do pracy z danymi geolokalizacyjnymi podaje jako pierwszą długość geograficzną, a następnie szerokość, ponieważ to odpowiada konwencji  $(x, y)$  stosowanej w matematyce. Format GeoJSON stosuje konwencję (długość geograficzna, szerokość geograficzna). Jeżeli będziesz używać innego framework'a, bardzo ważne jest sprawdzenie stosowanej przez niego konwencji.

## Utworzenie listy trzęsień ziemi

Przede wszystkim trzeba przygotować listę zawierającą wszystkie informacje o każdym odnotowanym trzęsieniu ziemi.

*Plik eq\_explore\_data.py:*

---

```
from pathlib import Path
import json

# Odczytanie danych jako ciągu tekstuowego i ich konwersja na obiekt Pythona.
path = Path('eq_data/eq_data_1_day_m1.geojson')
contents = path.read_text()
all_eq_data = json.loads(contents)

# Analiza wszystkich trzęsień na podstawie zbioru danych.
all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

---

Pobieramy dane powiązane z kluczem 'features' i umieszczamy je w słowniku all\_eq\_dicts. Wiemy, że plik zawiera informacje o 160 odnotowanych trzęsieniach ziemi, a dane wyjściowe programu potwierdzają wychwytcenie informacji o wszystkich zdarzeniach.

---

160

---

Zwróć uwagę na to, jak krótki jest kod programu. Elegancko sformatowany plik *readable\_eq\_data.json* ma ponad 6000 wierszy. Program składający się zaledwie kilku wierszy potrafi odczytać wszystkie te dane i umieścić je w słowniku Pythona. Następnym zadaniem jest pobranie informacji o sile każdego z odnotowanych trzęsień ziemi.

## Wyodrębnienie siły trzęsienia ziemi

Wykorzystując listę zawierającą dane o wszystkich trzęsieniach ziemi, można przeprowadzić iterację przez tę listę i wyodrębnić wszelkie interesujące nas informacje. Spójrz, jak wygląda pobranie danych o sile poszczególnych trzęsień ziemi.

*Plik eq\_explore\_data.py:*

---

```
--cięcie--
all_eq_dicts = all_eq_data['features']

mags = [] ①
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag'] ②
```

```
mags.append(mag)  
print(mags[:10])
```

---

Tworzymy pustą listę przeznaczoną do przechowywania informacji o trzęsieńach ziemi, a następnie przeprowadzamy iterację przez słownik `all_eq_dicts` (patrz wiersz ①). W pętli każde trzęsienie ziemi jest przedstawiane za pomocą słownika `eq_dict`. Natomiast siła danego trzęsienia ziemi jest przechowywana w sekcji "properties" słownika `eq_dict`, pod kluczem "mag" (patrz wiersz ②). Informacje o sile trzęsienia ziemi zapisujemy w zmiennej `mag`, której wartość jest następnie umieszczana na liście `mags`.

Wyświetlamy pierwsze dziesięć elementów listy, aby sprawdzić, czy na pewno mamy prawidłowe dane.

---

```
[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]
```

---

Kolejnym krokiem jest pobranie danych o miejscu wystąpienia poszczególnych trzęsień ziemi, co pozwoli przygotować ich mapę.

## Wyodrębnienie danych o miejscu wystąpienia trzęsienia ziemi

Dane dotyczące miejsca wystąpienia trzęsienia ziemi znajdują się w kluczu "geometry". W tym słowniku mamy klucz "coordinates", którego dwie pierwsze wartości na liście to długość i szerokość geograficzna. Spójrz, jak wygląda pobranie tych danych.

*Plik eq\_explore\_data.py:*

---

```
--cięcie--  
all_eq_dicts = all_eq_data['features']  
  
mags, lons, lats = [], [], []  
for eq_dict in all_eq_dicts:  
    mag = eq_dict['properties']['mag']  
    lon = eq_dict['geometry']['coordinates'][0] ❶  
    lat = eq_dict['geometry']['coordinates'][1]  
    mags.append(mag)  
    lons.append(lon)  
    lats.append(lat)  
  
print(mags[:10])  
print(lons[:5])  
print(lats[:5])
```

---

Tworzymy pustą listę przeznaczoną do przechowywania informacji o długości i szerokości geograficznej. Polecenie `eq_dict['geometry']` pozwala uzyskać

dostęp do słownika przedstawiającego element `geometry` dla danego trzęsienia ziemi (patrz wiersz ❶). Drugi klucz, 'coordinates', pobiera listę wartości powiązanych z tym kluczem. Za pomocą indeksu 0 pobieramy z listy współrzędnych geograficznych pierwszą wartość, która odpowiada długości geograficznej.

Wyświetlamy pierwsze pięć współrzędnych geograficznych (długość i szerokość), aby sprawdzić, czy na pewno mamy prawidłowe dane.

```
[1.6, 1.6, 2.2, 3.7, 2.92000008, 1.4, 4.6, 4.5, 1.9, 1.8]  
[-150.7585, -153.4716, -148.7531, -159.6267, -155.248336791992]  
[61.7591, 59.3152, 63.1633, 54.5612, 18.7551670074463]
```

Mając te dane, można przystąpić do ich przedstawienia na mapie.

## Budowanie mapy świata

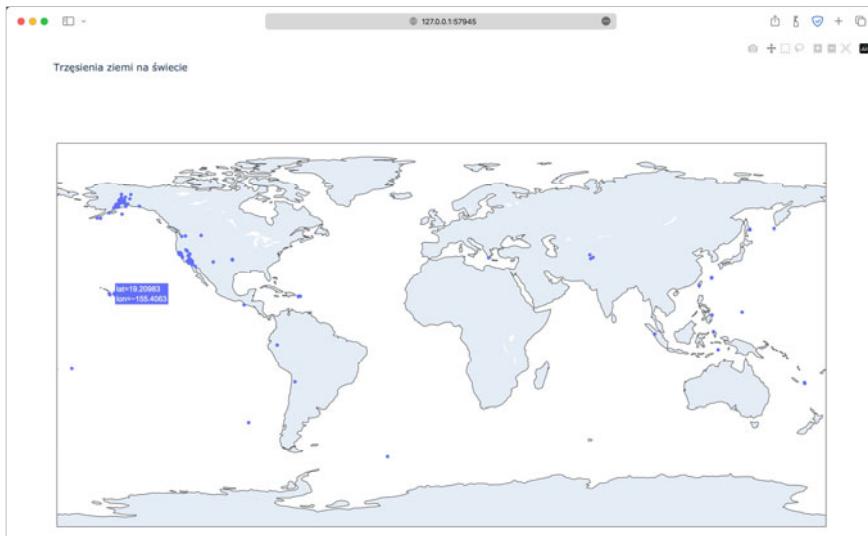
Wykorzystując zebrane dane, możemy stosunkowo łatwo utworzyć mapę świata. Wprawdzie nie będzie jeszcze wyglądała zbyt reprezentacyjnie, ale chcemy zanim zaczniemy koncentrować się na kwestiach związanych ze stylem i prezentacją, chcemy mieć pewność, że informacje wyświetlane są prawidłowo. Oto kod tworzący początkową wersję mapy.

Plik `eq_world_map.py`:

```
from pathlib import Path  
import json  
  
import plotly.express as px  
  
--cięcie--  
for eq_dict in all_eq_dicts:  
    --cięcie--  
  
    title = 'Trzęsienia ziemi na świecie'  
    fig = px.scatter_geo(lat=lats, lon=lons, title=title) ❶  
    fig.show()
```

Importujemy moduł `plotly.express` jako alias `px`, podobnie jak w rozdziale 15. Funkcja `scatter_geo()` (patrz wiersz ❶) pozwala umieścić punkty danych geograficznych na mapie. W najprostszym przypadku użycia takiego rodzaju wykresu konieczne jest dostarczenie jedynie listy długości geograficznych i listy szerokości geograficznych. Lista `lats` jest przekazywana argumentowi `lat`, natomiast lista `lons` argumentowi `lon`.

Po uruchomieniu programu powinieneś zobaczyć mapę pokazaną na rysunku 16.7. To ponownie pokazuje potężne możliwości biblioteki `plotly express`. Za pomocą zaledwie trzech wierszy kodu udało się przygotować mapę globalnej aktywności sejsmicznej na świecie.



Rysunek 16.7. Prosta mapa pokazująca wszystkie trzęsienia ziemi odnotowane w ciągu ostatnich 24 godzin

Można wprowadzić wiele modyfikacji, aby ta mapa była znacznie bardziej zrozumiała i łatwiejsza w odczycie. Przedstawię kilka takich zmian.

## Przedstawienie siły trzęsienia ziemi

Mapa trzęsień ziemi powinna pokazywać także ich siłę. Można więc uwzględnić jeszcze więcej informacji, ponieważ wiemy, że dane są poprawnie przedstawiane na wykresie.

---

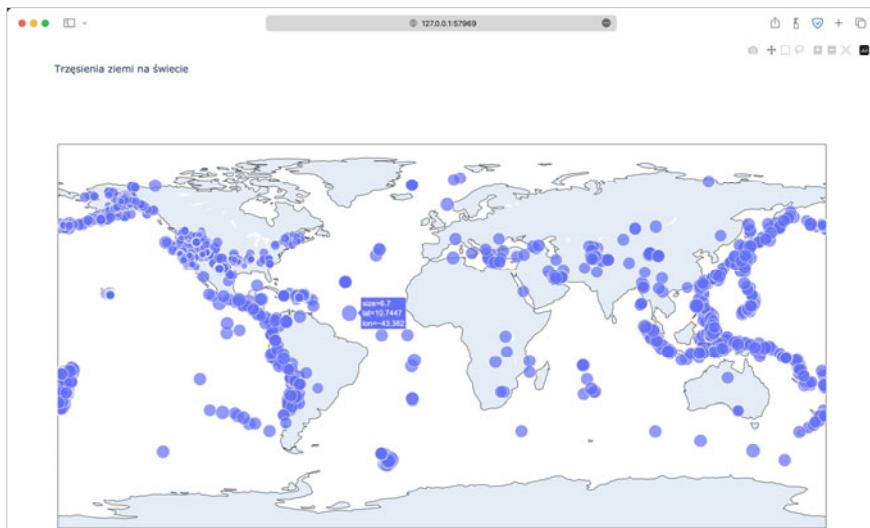
```
--cięcie--
# Odczytanie danych jako ciągu tekstuowego i ich konwersja na obiekt Pythona.
path = Path('eq_data/eq_data_30_day_m1.geojson')
contents = path.read_text()
--cięcie--

title = 'Trzęsienia ziemi na świecie'
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title)
fig.show()
```

---

Wczytujemy plik *eq\_data\_30\_day\_m1.geojson* w celu użycia danych przedstawiających trzęsienia ziemi w ostatnich 30 dniach. Użyty został także argument `size` wywołania `px.scatter_geo()`, który pozwala określić wielkość poszczególnych punktów na mapie. Wartością tego klucza jest lista `mags`, więc trzęsienia ziemi o większej sile zostaną przedstawione w postaci większych punktów na mapie.

Po uruchomieniu tego programu powinieneś zobaczyć mapę, którą pokazalem na rysunku 16.8. Trzęsienia ziemi występują zwykle na styku płyt tektonicznych, co odpowiada danym pokazanym na mapie.



Rysunek 16.8. Mapa pokazuje teraz siłę poszczególnych trzęsień ziemi

Wprawdzie ta mapa jest lepsza, ale wciąż trudno jest określić, które punkty reprezentują najsielniejsze trzęsienia ziemi. Mapę można więc jeszcze poprawić poprzez użycie koloru do pokazania siły trzęsienia ziemi.

## Dostosowanie koloru punktu

Oferowane przez plotly skale kolorów pozwalają dostosować do własnych potrzeb kolory poszczególnych znaczników, zgodnie z siłą trzęsień ziemi. Ponadto użyjemy innej projekcji dla mapy bazowej.

Plik eq\_world\_map.py:

```
--cięcie--  
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,  
                     color=mags, ❶  
                     color_continuous_scale='Viridis', ❷  
                     labels={'color':'Siła'}, ❸  
                     projection='natural earth', ❹  
)  
fig.show()
```

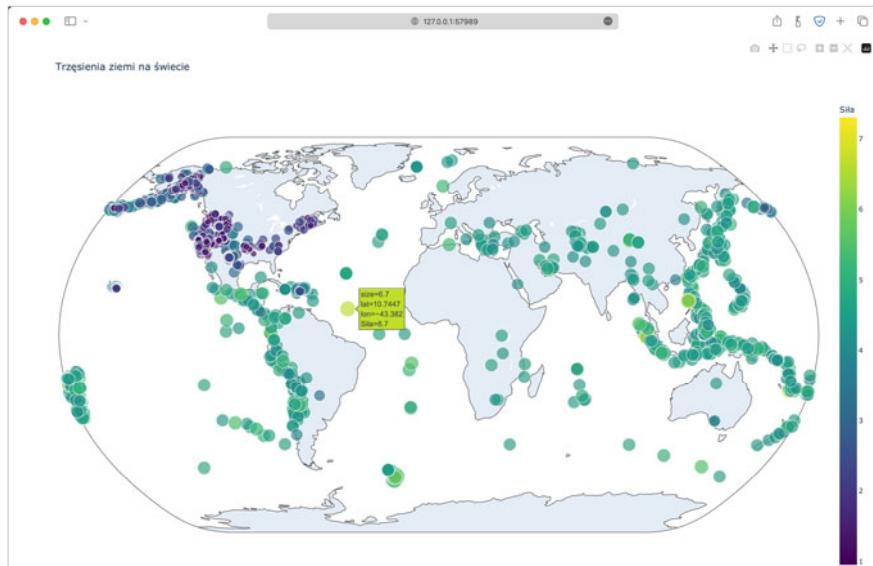
Wszystkie istotne zmiany zostały wprowadzone w wywołaniu funkcji px.  
→ `scatter_geo()`. Ustawienie 'color' wskazuje plotly wartości, które powinny być używane podczas ustalania, czy dany punkt mieści się na skali kolorów (patrz

wiersz ①). Lista mags została użyta do ustalenia wykorzystanego koloru, podobnie jak to miało miejsce w przypadku argumentu `size`.

Argument `color_continuous_scale` wskazuje ploty zakres kolorów do użycia (patrz wiersz ②): 'Viridis' to skala kolorów od ciemnoniebieskiego do jasnożółtego, sprawdza się ona doskonale w przypadku tego zbioru danych. Domyślnie skala kolorów po prawej stronie mapy jest oznaczona etykietą `color`. To nie od-daje znaczenia tych kolorów. Argument `labels` przedstawiony w rozdziale 15. pobiera wartość w postaci słownika (patrz wiersz ③). W omawianym przykładzie trzeba zdefiniować tylko jedną niestandardową etykietę na wykresie i upewnić się, że skala kolorów została zatytułowana *Sila* zamiast `color`.

W kodzie został dodany jeszcze jeden argument w celu zmiany mapy bazowej, na której zostały umieszczone dane dotyczące trzęsień ziemi. Argument `projection` akceptuje pewną liczbę powszechnie używanych projekcji mapy (patrz wiersz ④). Tutaj została użyta projekcja 'natural earth', która powoduje zaokrąglenie rogów mapy. Zwróć uwagę na przecinek po ostatnim argumencie. Gdy wywołanie funkcji ma długą listę argumentów obejmujących wiele wierszy, jak w omawianym przykładzie, wówczas powszechną praktyką jest umieszczenie przecinka na końcu wiersza. Dzięki temu wiadomo, że można dodać kolejny argument w następnym wierszu.

Po uruchomieniu tego programu powinieneś zobaczyć znacznie lepiej wygładającą mapę. Na rysunku 16.9 skala kolorów pokazuje siłę poszczególnych trzęsień ziemi. Najsilniejsze są przedstawione za pomocą jasnożółtych punktów, słabsze zaś za pomocą ciemniejszych. Na podstawie tych danych można stwierdzić, w których rejonach świata mamy do czynienia z większą aktywnością sejsmiczną ziemi.



Rysunek 16.9. Mapa pokazuje trzęsienia ziemi odnotowane w ciągu 30 dni. Kolor został użyty do przedstawienia siły wstrząsu

## Inne skale kolorów

Do dyspozycji masz wiele innych skali kolorów. Jeżeli chcesz poznać dostępne, w powłoce Pythona wydaj następujące dwa polecenia:

```
>>> import plotly.express as px  
>>> px.colors.named_colorscales()  
['aggrnyl', 'agsunset', 'blackbody', ..., 'mygbm']
```

Wypróbuj inne skale na mapie trzęsień ziemi bądź dla innego dowolnego zbioru danych, w którego przypadku ciągła zmiana kolorów może pomóc w dostrzeżeniu wzorców występujących w danych.

## Dodanie tekstu wyświetlanego po wskazaniu punktu na mapie

Aby dokończyć pracę z mapą, dodamy tekst wyświetlany po umieszczeniu kurSORA myszy na punkcie reprezentującym trzęsienie ziemi. Poza wyświetlaniem długości i szerokości geograficznej, które są danymi domyślnymi, informacje będą zawierały również siłę i przybliżoną lokalizację trzęsienia ziemi.

W celu wprowadzenia tej zmiany trzeba z pliku pobrać nieco więcej danych.

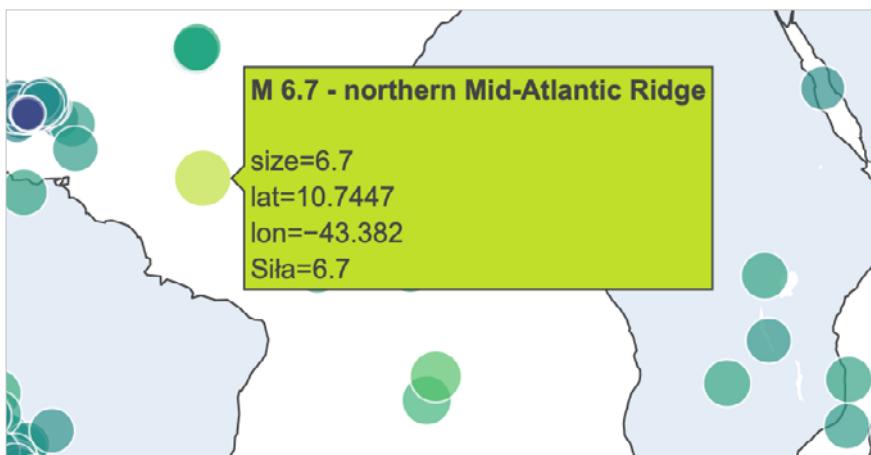
Plik eq\_world\_map.py:

```
--cięcie--  
mags, lons, lats, eq_titles = [], [], [], [] ❶  
for eq_dict in all_eq_dicts:  
    mag = eq_dict['properties']['mag']  
    lon = eq_dict['geometry']['coordinates'][0]  
    lat = eq_dict['geometry']['coordinates'][1]  
    eq_title = eq_dict['properties']['title'] ❷  
    mags.append(mag)  
    lons.append(lon)  
    lats.append(lat)  
    eq_titles.append(eq_title)  
  
title = 'Trzęsienia ziemi na świecie'  
fig = px.scatter_geo(lat=lats, lon=lons, size=mags, title=title,  
                     --cięcie--  
                     projection='natural earth',  
                     hover_name=eq_titles, ❸  
)  
fig.show()
```

Przede wszystkim tworzymy listę o nazwie eq\_titles, przeznaczoną do przechowywania etykiety każdego punktu (patrz wiersz ❶). Sekcja "title" danych trzęsienia ziemi zawiera opisową nazwę siły i miejsca wystąpienia, a także długość

i szerokość geograficzną. W wierszu ② pobieramy te informacje i przypisujemy zmiennej eq\_title, którą następnie umieszczamy na liście eq\_titles.

W wywołaniu px.scatter\_geo() przekazujemy listę eq\_titles do argumentu hover\_name (patrz wiersz ③). Plotly wykorzystuje te informacje jako etykietę dla poszczególnych punktów wyświetlana po umieszczeniu kurSORA myszy na punkcie. Po uruchomieniu programu, jeśli umieścisz kurSORa myszy na punkcie, powinieneś zobaczyć etykietę zawierającą informacje m.in. o miejscu wystąpienia danego trzęsienia ziemi i jego sile. Przykład takich informacji pokazałem na rysunku 16.10.



Rysunek 16.10. Umieszczenie kursora myszy na punkcie powoduje wyświetlenie informacji na temat danego trzęsienia ziemi.

To jest imponujące! W mniej niż 30 wierszach kodu utworzyliśmy wizualnie atrakcyjną i zawierającą czytelne informacje mapę globalnej aktywności sejsmicznej ziemi, która ilustruje również strukturę geologiczną naszej planety. Plotly oferuje znacznie więcej sposobów dostosowania wyglądu i sposobu działania wizualizacji. Wykorzystując dostępne opcje plotly, możesz tworzyć wykresy i mapy, które będą zawierały dokładnie to, co chcesz pokazać.

## ZRÓB TO SAM

**16.6. Refaktoryzacja.** Pętla pobierająca dane z all\_eq\_dicts używa zmiennych do przechowywania siły, długości i szerokości geograficznej oraz tytułu odnotowanego trzęsienia ziemi. Następnie te wartości są umieszczane na odpowiednich listach. Takie podejście wybrałem dla tego, że wyraźnie pokazuje, jak dane są pobierane z pliku GeoJSON. Jednak niekoniecznie musisz stosować takie rozwiązanie. Zamiast użycia tych zmiennych tymczasowych poszczególne wartości pobieraj z eq\_dict i dołączaj do odpowiedniej listy. To pozwoli skrócić kod pętli do zaledwie czterech wierszy.

**16.7. Zautomatyzowany tytuł.** W tym podrozdziale użyłem ogólnego tytułu *Trzęsienia ziemi na świecie*. Zamiast tego tytuły można pobierać ze zbioru danych, a dokładnie z metadanych pliku GeoJSON. Pobierz tę wartość i przypisz zmiennej `title`.

**16.8. Ostatnie trzęsienia ziemi.** W internecie możesz znaleźć pliki danych zawierające informacje o trzęsieniach ziemi odnotowanych w ciągu ostatniej godziny, ostatniego dnia oraz ostatnich 7 i 30 dni. Wystarczy przejść na stronę <https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php>, na której znajdują się łącza do zbiorów danych dla różnych przedziałów czasu. Te dane są podzielone według siły wstrząsu. Pobierz jeden z tych zbiorów danych, a następnie utwórz wizualizację przedstawiającą ostatnią aktywność sejsmiczną ziemi.

**16.9. Pożary na świecie.** W materiałach przygotowanych dla tego rozdziału znajduje się plik o nazwie *world\_fires\_1\_day.csv*. Zawiera on informacje o pożarach w różnych zakątkach świata, dla każdego pożaru jest podana długość i szerokość geograficzna oraz jego siła. Wykorzystując materiał przedstawiony w rozdziale, przygotuj mapę pokazującą, gdzie na świecie odnotowano pożary.

Najnowszą wersję wymienionego pliku możesz pobrać ze strony <https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data>. Dostępne łącza prowadzą do danych zapisanych w formacie CSV, które znajdziesz w sekcji *SHP, KML, and TXT Files*.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak pracować z rzeczywistymi zbiorami danych. Zobaczyłeś, jak można przetwarzać pliki w formatach CSV i GeoJSON oraz jak wyodrębniać interesujące Cię dane, na których chcesz się skoncentrować. Na podstawie historycznych danych pogodowych doskonaliłeś sposób pracy z biblioteką matplotlib, nauczyłeś się używać modułu datet im e i umieszczać na jednym wykresie wiele serii danych. Dowiedziałeś się również, jak wygenerować mapę świata za pomocą plotly, a także jak nadawać style mapom i wykresom tworzonym przez plotly.

Gdy zdobędziesz jeszcze większe doświadczenie w pracy z plikami w formatach CSV i JSON, będziesz umiał przetwarzać niemal wszystkie rodzaje danych, jakie chciałbyś analizować. Większość dostępnych w internecie zbiorów danych można pobierać w dowolnym z tych dwóch wymienionych formatów, lub nawet w obu. Oczywiście po opanowaniu pracy z formatami CSV i JSON będziesz mógł poznawać sposobu pracy także z innymi formatami danych.

W następnym rozdziale utworzymy programy automatycznie zbierające dane ze źródeł internetowych, a następnie przygotujemy wizualizacje na podstawie tych danych. Tego rodzaju umiejętności mogą być przydatne, gdy traktujesz programowanie jako hobby, ale jednocześnie mają istotne znaczenie, gdy chcesz profesjonalnie zająć się programowaniem.

# 17

## Praca z API



W TYM ROZDZIALE DOWIESZ SIĘ, JAK UTWORZYĆ SAMODZIELNIE DZIAŁAJĄCY PROGRAM PRZEZNACZONY DO GENEROWANIA WIZUALIZACJI NA PODSTAWIE OTRZYMANYCH DANYCH. TWÓJ PROGRAM WYKORZYSTA INTERFEJS

*programowania aplikacji* (API), aby automatycznie wysyłać do witryny internetowej żądania dotyczące udostępnienia określonych informacji zamiast całych stron. Następnie te informacje zostaną wykorzystane do wygenerowania wizualizacji. Ponieważ tego rodzaju programy do generowania wizualizacji będą używaly najnowszych danych, wizualizacje te będą zawsze aktualne, nawet w przypadku szybko zmieniających się danych.

### Użycie API

API jest częścią witryny internetowej zaprojektowanej do współdziałania z programami, które stosują konkretne adresy URL, aby wysyłać żądania dotyczące udostępnienia określonych informacji. Tego rodzaju żądania są nazywane *wywołaniami API*. Żądane dane zostaną zwrócone w formacie łatwym do późniejszego przetworzenia, takim jak JSON i CSV. Większość aplikacji opierających swoje działanie na zewnętrznych źródłach danych, na przykład aplikacje zapewniające integrację z witrynami społecznościowymi, wykorzystuje wspomniane wywołania API.

## Git i GitHub

Dane do tworzonej wizualizacji będą pochodziły z witryny GitHub (<https://github.com/>), czyli serwisu pozwalającego programistom na wspólną pracę nad projektami. API oferowane przez GitHub wykorzystamy do pobrania informacji o projektach Pythona znajdujących się w tej witrynie, a następnie w ploty wygenerujemy interaktywną wizualizację względnej popularności tych projektów.

Nazwa serwisu GitHub pochodzi od Gita, czyli rozproszonego systemu kontroli wersji pozwalającego zespołom programistów współpracować przy projektach. Git pomaga użytkownikom zarządzać ich pracę w projekcie, aby zmiany wprowadzone przez jedną osobę nie kłociły się ze zmianami wprowadzanymi przez innych. Kiedy implementujesz nową funkcję w projekcie, Git śledzi zmiany wprowadzane w poszczególnych plikach. Kiedy nowy kod działa zgodnie z oczekiwaniemi, wówczas *zatwierdzasz* (ang. *commit*) wprowadzone zmiany, a Git rejestruje nowy stan projektu. Jeżeli popełnisz błąd i chcesz cofnąć wprowadzone zmiany, zawsze możesz bardzo łatwo powrócić do dowolnej z wcześniejszej działających wersji. (Aby dowiedzieć się więcej na temat kontroli wersji za pomocą Gita, zajrzyj do dodatku D). Projekty w serwisie GitHub są przechowywane w tak zwanych *repozytoriach*, które zawierają wszystko to, co jest powiązane z projektem: jego kod źródłowy, informacje o osobach pracujących nad projektem, wszelkie zgłoszenia błędów itd.

Kiedy użytkownik GitHub lubi dany projekt, może oznaczyć go „gwiazdką”, aby w ten sposób okazać swoje wsparcie i monitorować rozwój programów, które być może będzie chciał kiedyś wykorzystać. W tym rozdziale utworzymy program automatycznie pobierający informacje o najczęściej oznaczanych gwiazdką projektach Pythona w GitHub, a następnie na podstawie tych danych przygotujemy wizualizacje.

## Żądanie danych za pomocą wywołania API

API serwisu GitHub pozwala na żądanie wielu różnych informacji za pomocą oferowanych wywołań API. Aby zobaczyć, jak może wyglądać wywołanie API, w pasku adresu przeglądarki internetowej wpisz poniższy adres, a następnie naciśnij klawisz *Enter*:

---

<https://api.github.com/search/repositories?q=language:python+sort:stars>

---

Powyższe wywołanie zwróci liczbę projektów Pythona aktualnie znajdujących się w serwisie GitHub oraz informacje o najpopularniejszych repozytoriach Pythona. Przeanalizujmy przedstawione wywołanie. Część pierwsza, <https://api.github.com/>, kieruje żądanie do tego fragmentu witryny GitHub, który jest odpowiedzialny za obsługę wywołań API. Kolejna część, `search/repositories`, nakazuje API przeprowadzenie wyszukiwania wszystkich repozytoriów w serwisie GitHub.

Znak zapytania po `repositories` sygnalizuje, że zostanie przekazany argument. Litera `q` oznacza zapytanie (*query*), a znak równości pozwala na rozpoczęcie powadania zapytania (`=`). Używając `language:python`, wskazujemy, że jesteśmy

zainteresowani jedynie informacjami o repozytoriach, w których głównym językiem programowania jest Python. Ostatnia część, `+sort:stars`, powoduje sortowanie projektów według liczby gwiazdek, którymi zostały oznaczone.

Poniższy fragmentu kodu pokazuje kilka pierwszych wierszy otrzymanej odpowiedzi.

```
{  
    "total_count": 8961993, ❶  
    "incomplete_results": true, ❷  
    "items": [ ❸  
        {  
            "id": 54346799,  
            "node_id": "MDEwOlJlcG9zaXRvcnk1NDM0Njc50Q==",  
            "name": "public-apis",  
            "full_name": "public-apis/public-apis",  
            --cięcie--  
        }  
    ]  
}
```

Jak widzisz, odpowiedź pochodząca z omówionego adresu URL nie jest przeznaczona do bezpośredniego wykorzystania przez użytkowników, ponieważ została udzielona w formacie, który jest przetwarzany przez program. W chwili powstawania książki GitHub zawiera ponad 9 milionów projektów, w których głównym językiem programowania jest Python (patrz wiersz ❶). Ponieważ wartość `"incomplete_results"` wynosi `true` (patrz wiersz ❷), wiemy, że serwis GitHub nie przetworzył w pełni tego żądania. GitHub ogranicza czas wykonywania żądania, aby zapewnić dostępność API dla wszystkich użytkowników. W omawianym przykładzie zapytanie znalazło część najpopularniejszych repozytoriów Pythona, ale nie miało czasu na znalezienie wszystkich. Ten problem wkrótce usuniemy. Zwrócony element `"items"` to po prostu lista zawierająca szczegółowe informacje o najpopularniejszych projektach tworzonych w Pythonie w serwisie GitHub (patrz wiersz ❸).

## Instalacja requests

Pakiet `requests` pozwala programowi utworzonemu w Pythonie w łatwy sposób wysyłać żądanie dotyczące udostępnienia informacji z witryny internetowej oraz analizować otrzymaną odpowiedź. W celu instalacji tego pakietu należy użyć menedżera pakietów pip.

```
$ python -m pip install --user requests
```

Jeżeli do uruchamiania programów lub sesji Pythona w powłoce używasz polecenia `python3` lub innego, musisz odpowiednio zmodyfikować przedstawione wcześniej polecenie:

```
$ python3 -m pip install --user requests
```

## Przetworzenie odpowiedzi API

Przystępujemy teraz do utworzenia programu wykonującego wywołania API i przetwarzającego otrzymaną odpowiedź.

Plik `python_repos.py`:

---

```
import requests

# Wykonanie wywołania API i zachowanie otrzymanej odpowiedzi.
url = "https://api.github.com/search/repositories" ❶
url += "?q=language:python+sort:stars+stars:>10000"

headers = {'Accept': 'application/vnd.github.v3+json'} ❷
r = requests.get(url, headers=headers) ❸
print(f"Kod stanu: {r.status_code}") ❹

# Konwersja obiektu odpowiedzi na słownik.
response_dict = r.json() ❺

# Przetworzenie wyników.
print(response_dict.keys())
```

---

Na początku importujemy moduł `requests`. W wierszu ❶ przechowujemy adres URL wywołania API w zmiennej o nazwie `url`. Ponieważ jest to długий adres URL, został podzielony na dwa wiersze. Pierwszy zawiera część główną adresu URL, natomiast w drugim znajdują się ciąg tekstowy zapytania. Względem wcześniejszego zapytania tutaj mamy warunek dodatkowy, `stars:>10000`, który powoduje wyszukiwanie w serwisie GitHub jedynie repozytoriów projektów Pythona mających więcej niż 10 000 gwiazdek. To powinno pozwolić na zwrócenie pełnego i spójnego zbioru wynikowego.

Aktualnie GitHub korzysta z trzeciej wersji API, więc używaną wersję API trzeba zdefiniować w nagłówkach wywołania. Pozwala to żądać użycia wskazanej wersji API i otrzymać wyniki w formacie JSON (patrz wiersz ❷). Następnie wykorzystujemy `requests` w celu wykonania wywołania do API ❸. Wywołujemy metodę `get()` i przekazujemy do niej adres URL oraz zdefiniowany wcześniej nagłówek. Odpowiedź otrzymaną na żądanie przechowujemy w zmiennej `r`.

Obiekt odpowiedzi zawiera atrybut o nazwie `status_code`, który wskazuje, czy wykonanie żądania zakończyło się powodzeniem. (Kod stanu 200 wskazuje na żądanie zakończone sukcesem). W wierszu ❹ wyświetlamy wartość `status_code`, aby mieć pewność o prawidłowym wykonaniu wywołania API. Odpowiedź na wywołanie API zawiera informacje w formacie JSON. Dlatego też w wierszu ❺ używamy metody `json()` w celu konwersji tych informacji do postaci słownika Pythona. Otrzymany słownik jest przechowywany w zmiennej `response_dict`.

Na koniec wyświetlamy klucze ze słownika `response_dict` i otrzymujemy późniejsze dane wyjściowe:

---

```
Kod stanu: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

---

Ponieważ kod stanu wynosi 200, wiemy, że wykonanie żądania zakończyło się powodzeniem. Słownik odpowiedzi zawiera tylko trzy klucze: 'total\_count', 'incomplete\_results' i 'items'. Przyjrzymy się teraz dokładniej temu słownikowi odpowiedzi.

## Praca ze słownikiem odpowiedzi

Skoro mamy informacje otrzymane z wywołania API i przechowywane w postaci słownika, możemy przystąpić do pracy z tymi danymi. Zaczniemy od wygenerowania pewnego rodzaju podsumowania otrzymanych informacji. To jest dobry sposób na upewnienie się, że otrzymane informacje są zgodne z oczekiwanyimi. W kolejnym kroku będziemy mogli rozpocząć analizę interesujących nas danych.

*Plik python\_repos.py:*

---

```
import requests

# Wykonanie wywołania API i zachowanie otrzymanej odpowiedzi.
--cięcie--

# Konwersja obiektu odpowiedzi na słownik.
response_dict = r.json()
print(f"Całkowita liczba repozytoriów: {response_dict['total_count']}") ❶
print(f"Pełny zbiór wynikowy?: {not response_dict['incomplete_results']}") ❷

# Przetworzenie informacji o repozytoriach.
repo_dicts = response_dict['items'] ❸
print(f'Liczba zwróconych repozytoriów: {len(repo_dicts)}')

# Przeanalizowanie pierwszego repozytorium.
repo_dict = repo_dicts[0] ❹
print(f'\nKlucz: {len(repo_dict)}') ❺
for key in sorted(repo_dict.keys()):
    print(key)
```

---

W wierszu ❶ wyświetlamy wartość przypisaną kluczowi 'total\_count', która określa całkowitą liczbę repozytoriów Pythona zwróconych przez to wywołanie API. Użyta została także wartość powiązana z kluczem 'incomplete\_results', aby było wiadomo, że serwis GitHub w pełni przetworzył to zapytanie. Zamiast bezpośrednio wyświetlać wartość tego klucza, została wyświetlona inna wartość: True będzie wskazywała na otrzymanie pełnego zbioru wynikowego.

Wartość przypisana kluczowi 'items' to lista zawierająca wiele słowników, z których każdy przechowuje informacje o jednym repozytorium Pythona. W wierszu ❸ umieszczaśmy tę listę słowników w zmiennej `repo_dicts`. Następnie wyświetlamy wielkość listy `repo_dicts`, aby dowiedzieć się, o ilu repozytoriach otrzymaliśmy informacje.

Aby dokładniej przyjrzeć się informacjom dotyczącym poszczególnych repozytoriów, z listy `repo_dicts` pobieramy pierwszy element i umieszczać go w `repo_dict` (patrz wiersz ❸). Wyświetlamy liczbę kluczy w słowniku, aby sprawdzić ilość otrzymanych informacji (patrz wiersz ❹). Następnie w wierszu ❺ wyświetlamy wszystkie klucze słownika, co pozwala nam się zorientować, z jakiego rodzaju informacjami będziemy mieli do czynienia.

Wygenerowane dane wyjściowe dają nam ogólne wyobrażenie rzeczywistych danych:

---

```
Kod stanu: 200
Całkowita liczba repozytoriów: 248 ❶
Pełny zbiór wynikowy?: True ❷
Liczba zwróconych repozytoriów: 30

Klucze: 78 ❸
allow_forking
archive_url
archived
--cięcie--
url
visibility
watchers
watchers_count
```

---

W chwili powstawania książki w serwisie GitHub znajdowało się 248 repozytoriów Pythona mających ponad 10 000 gwiazdek (patrz wiersz ❶). Jak widać w wierszu ❷, serwis GitHub był w stanie w pełni przetworzyć to wywołanie API. W odpowiedzi na nie zostały przekazane informacje o pierwszych 30 repozytoriach dopasowanych do warunku zapytania. Jeżeli chcesz otrzymać informacje o kolejnych repozytoriach, możesz zażądać następnych stron danych.

API GitHub dostarcza wielu informacji o poszczególnych repozytoriach, na co wskazuje 78 kluczy znajdujących się w słowniku `repo_dict`, jak możesz zobaczyć w wierszu ❸. Kiedy przejrzyesz te klucze, będziesz wiedzieć, jakiego rodzaju informacje można otrzymać o każdym projekcie. (Jedyny sposób ustalenia, jakiego rodzaju informacje są dostępne za pomocą API, to zapoznanie się z dokumentacją, lub też przeanalizowanie informacji przetworzonych przez kod, jak ma to miejsce w omawianym przypadku).

Pobierzemy teraz wartości pewnych kluczy ze słownika `repo_dict`.

*Plik `python_repos.py`:*

---

```
--cięcie--
# Przeanalizowanie pierwszego repozytorium.
repo_dict = repo_dicts[0]

print("\nWybrane informacje o pierwszym repozytorium:")
print(f"Nazwa: {repo_dict['name']}") ❶
```

```
print(f"Właściciel: {repo_dict['owner']['login']}") ❷
print(f"Gwiazdki: {repo_dict['stargazers_count']}") ❸
print(f"Repozytorium: {repo_dict['html_url']}")  

print(f"Utworzzone: {repo_dict['created_at']}") ❹
print(f"Uaktualnione: {repo_dict['updated_at']}") ❺
print(f"Opis: {repo_dict['description']}")
```

---

W powyższym kodzie umieściliśmy polecenia przeznaczone do wyświetlenia wartości kilku wybranych kluczy ze słownika informacji o repozytorium. Polecenie znajdujące się w wierszu ❶ wyświetli nazwę projektu. Dane dotyczące właściciela projektu zostały umieszczone w słowniku, więc w wierszu ❷ używamy klucza owner, aby uzyskać dostęp do tego słownika, a następnie za pomocą klucza login pobieramy nazwę użytkownika, którą właściciel stosuje podczas logowania. W wierszu ❸ wyświetlamy liczbę gwiazdek zebranych przez dany projekt oraz adres URL danego projektu w repozytorium GitHub. Następne polecenia wyświetlają datę utworzenia repozytorium (patrz wiersz ❹) oraz datę jego ostatniego uaktualnienia (patrz wiersz ❺). Na końcu wyświetlamy opis repozytorium.

Ostatecznie wygenerowane dane wyjściowe będą miały postać podobną do przedstawionej poniżej:

---

```
Kod stanu: 200
Całkowita liczba repozytoriów: 248
Pełny zbiór wynikowy?: True
Liczba zwróconych repozytoriów: 30

Wybrane informacje o pierwszym repozytorium:
Nazwa: public-apis
Właściciel: public-apis
Gwiazdki: 191493
Repozytorium: https://github.com/public-apis/public-apis
Utworzzone: 2016-03-20T23:49:42Z
Uaktualnione: 2022-05-12T06:37:11Z
Opis: A collective list of free APIs
```

---

Jak widzisz, w czasie, kiedy powstawała ta książka, w serwisie GitHub projektem wykorzystującym Pythona, który zebrał najwięcej gwiazdek, był *public-apis*. Właścicielem tego projektu jest organizacja o takiej samej nazwie, a sam projekt został oznaczony gwiazdką przez prawie 200 tysięcy użytkowników serwisu GitHub. Wygenerowane dane wyjściowe zawierają adres URL repozytorium projektu, jego datę utworzenia (marzec 2016) oraz datę ostatniej aktualizacji (maj 2022). Na końcu z opisu dowiadujemy się, że to repozytorium zawiera listę bezpłatnych API, które mogą zainteresować programistów.

## Podsumowanie repozytoriów najczęściej oznaczanych gwiazdką

Podeczas tworzenia wizualizacji dla tych danych chcemy uwzględnić więcej niż tylko jedno repozytorium. Przygotujemy więc pętlę wyświetlającą wybrane informacje o wszystkich repozytoriach zwróconych przez wywołanie API, co pozwoli nam na ich uwzględnienie w tworzonej wizualizacji.

Plik `python_repos.py`:

```
--cięcie--  
# Przetworzenie informacji o repozytoriach.  
repo_dicts = response_dict['items']  
print(f"Liczba zwróconych repozytoriów: {len(repo_dicts)}")  
  
print("\nWybrane informacje o każdym repozytorium:") ❶  
for repo_dict in repo_dicts: ❷  
    print(f"\nNazwa: {repo_dict['name']}")  
    print(f"Właściciel: {repo_dict['owner']['login']}")  
    print(f"Gwiazdki: {repo_dict['stargazers_count']}")  
    print(f"Repozytorium: {repo_dict['html_url']}")  
    print(f"Opis: {repo_dict['description']}")
```

W wierszu ❶ zdefiniowaliśmy komunikat początkowy. Dalej w wierszu ❷ rozpoczęta się pętla odpowiedzialna za przeprowadzenie iteracji przez wszystkie słowniki znajdujące się w `repo_dicts`. W bloku kodu pętli wyświetlamy nazwę każdego projektu, jego właściciela, liczbę otrzymanych gwiazdek, adres URL repozytorium w serwisie GitHub oraz opis projektu:

```
Kod stanu: 200  
Całkowita liczba repozytoriów: 3494040  
Liczba zwróconych repozytoriów: 30
```

Wybrane informacje o każdym repozytorium:

```
Nazwa: public-apis  
Właściciel: public-apis  
Gwiazdki: 191493  
Repozytorium: https://github.com/public-apis/public-apis  
Opis: A collective list of free APIs
```

```
Nazwa: system-design-primer  
Właściciel: donnemartin  
Gwiazdki: 179952  
Repozytorium: https://github.com/donnemartin/system-design-primer  
Opis: Learn how to design large-scale systems. Prep for the system  
design interview. Includes Anki flashcards.  
--cięcie--
```

Nazwa: PayloadsAllTheThings  
Właściciel: swisskyrepo  
Gwiazdki: 37227  
Repozytorium: <https://github.com/swisskyrepo/PayloadsAllTheThings>  
Opis: A list of useful payloads and bypass for Web Application Security and Pentest/CTF

---

W wynikach pojawiają się pewne interesujące projekty i naprawdę warto spojrzeć na kilka z nich. Jednak nie poświęcaj na to zbyt wiele czasu, ponieważ za chwilę przystąpimy do utworzenia wizualizacji, która znacznie ułatwi nam przeglądanie wygenerowanych wyników.

## Monitorowanie ograniczeń liczby wywołań API

Większość API stosuje pewnego rodzaju *ograniczenia liczby wywołań*, czyli definiuje maksymalną liczbę żądań, jakie można wykonać w określonym przedziale czasu. Jeżeli chcesz sprawdzić, czy osiągnąłeś limit ustalony przez serwis GitHub, w przeglądarce internetowej wpisz adres [https://api.github.com/rate\\_limit](https://api.github.com/rate_limit). Powinieneś otrzymać dane wyjściowe podobne do przedstawionych poniżej.

```
{  
  "resources": {  
    --cięcie--  
    "search": { ❶  
      "limit": 10, ❷  
      "remaining": 9, ❸  
      "reset": 1652338832 ❹  
      "used": 1,  
      "resource": "search"  
    },  
    --cięcie--
```

---

Interesujące nas informacje dotyczą liczby dozwolonych operacji wyszukiwania API (patrz wiersz ❶). Jak możesz zobaczyć w wierszu ❷, ten limit wynosi 10 żądań na minutę i pozostało nam jeszcze 9 do wykonania w bieżącej minucie (patrz wiersz ❸). Wartość klucza reset przedstawia wyrażoną jako *czas epoki systemu UNIX* (czyli liczba sekund, które upłynęły od północy 1 stycznia 1970 roku) godzinę, kiedy nastąpi wyzerowanie limitu (patrz wiersz ❹). Po wykorzystaniu przydzielonego limitu otrzymasz krótką odpowiedź informującą o tym. Wystarczy wówczas chwilę odczekać na wyzerowanie limitu żądań.

**UWAGA** W przypadku wielu API wymagana jest rejestracja, a po jej dokonaniu można otrzymać klucz API pozwalający na wykonywanie wywołań API. W trakcie powstania tej książki serwis GitHub nie wymagał żadnej rejestracji. Jednak gdy użyjemy klucza API, przydzielony nam limit wywołań będzie znacznie większy.

# Wizualizacja repozytoriów za pomocą pakietu plotly

Skoro zebraliśmy pewne interesujące nas dane, możemy przejść do przygotowania wizualizacji pokazującej względną popularność projektów Pythona w serwisie GitHub. W tym podrozdziale utworzymy interaktywny wykres słupkowy. Wysokość poszczególnych słupków będzie odzwierciedlać liczbę gwiazdek zebranych przez projekt. Kliknięcie słupka spowoduje przejście na stronę danego projektu w serwisie GitHub.

Kopię tworzonego w tym rozdziale programu zapisz pod nazwą `python_repos_visual.py`, a następnie zmodyfikuj go w pokazany tutaj sposób.

Plik `python_repos_visual.py`:

---

```
import requests
import plotly.express as px

# Wykonanie wywołania API i zachowanie otrzymanej odpowiedzi.
url = "https://api.github.com/search/repositories"
url += "?q=language:python+sort:stars+stars:>10000"

headers = {'Accept': 'application/vnd.github.v3+json'}
r = requests.get(url, headers=headers)
print(f"Kod stanu: {r.status_code}") ❶

# Przetworzenie otrzymanych wyników.
response_dict = r.json()
print(f"Pełny zbiór wynikowy?: {not response_dict['incomplete_results']}") ❷

# Przetworzenie informacji o repozytoriach.
repo_dicts = response_dict['items']
repo_names, stars = [], [] ❸
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

# Utworzenie wizualizacji.
fig = px.bar(x=repo_names, y=stars) ❹
fig.show()
```

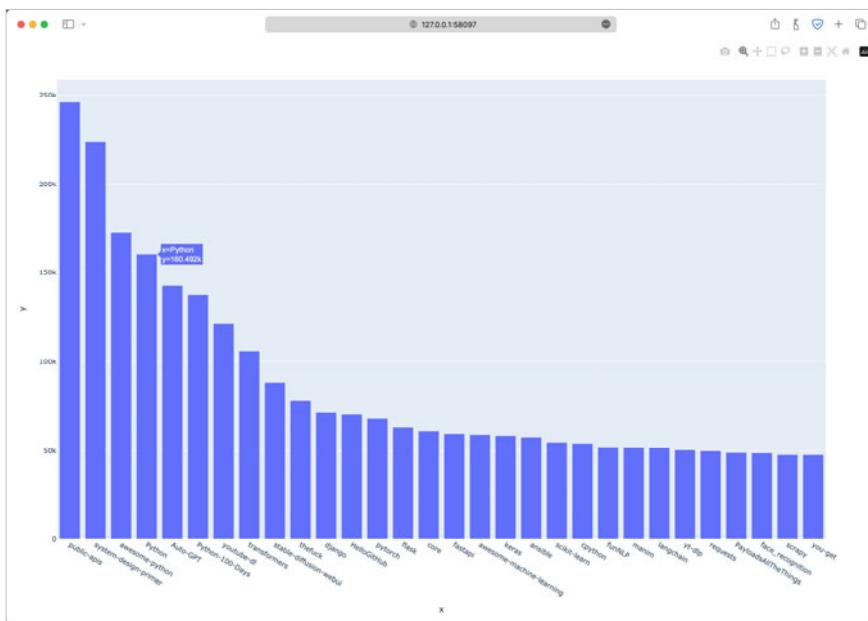
---

Importujemy `plotly express` i wykonujemy zapytanie API, podobnie jak wcześniej. Nadal wyświetlamy kod stanu odpowiedzi na wywołanie API. Dzięki temu będziemy wiedzieć, gdy wystąpi jakikolwiek problem związany z wywołaniem API (patrz wiersz ❶). Podczas przetwarzania wyników nadal będzie wyświetlony komunikat wskazujący, czy otrzymaliśmy pełny zbiór wynikowy. (patrz wiersz ❷). Usuwamy trochę kodu odpowiedzialnego za przetworzenie odpowiedzi API, ponieważ wyszliśmy z fazy analizy danych i dokładnie wiemy, jakie informacje będą nam potrzebne.

W wierszu ❸ tworzymy dwie puste listy przeznaczone do przechowywania danych uwzględnionych na wykresie. Potrzebujemy nazwy projektu, aby wygenerować etykiety dla poszczególnych słupków (`repo_names`). Pobieramy także liczbę gwiazdek, by określić wysokość słupków (`stars`). W pętli do przygotowanych wcześniej list dodajemy nazwę projektu i liczbę zebranych przez niego gwiazdek.

Do utworzenia początkowej wizualizacji potrzeba zaledwie dwóch wierszy kodu (patrz wiersz ❹). Jest to spójne z filozofią plotly express, zgodnie z którą wizualizacja powinna być dostępna jak najszybciej, a później można się zająć jej dopracowaniem. W omawianym przykładzie funkcja `px.bar()` została użyta do utworzenia wykresu słupkowego. Przekazujemy jej listę `repo_names` jako argument `x` oraz listę `stars` jako argument `y`.

Wygenerowany wykres pokazałem na rysunku 17.1. Jak widzisz, kilka pierwszych projektów cieszy się zdecydowanie większą popularnością od pozostałych. Jednak wszystkie pokazane na wykresie projekty są bardzo ważne w ekosystemie Pythona.



Rysunek 17.1. Oznaczone największą liczbą gwiazdek projekty Pythona w serwisie GitHub

## Nadawanie stylu wykresowi

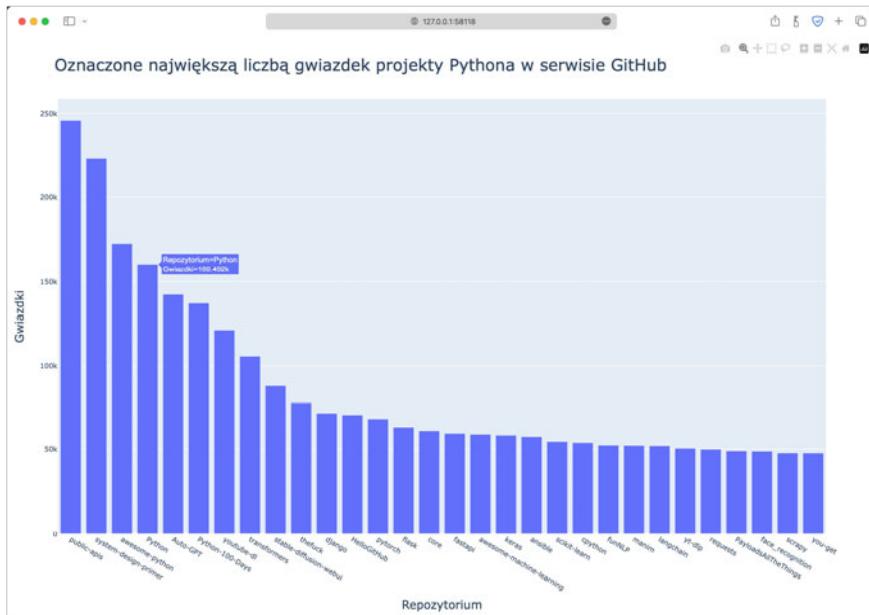
Plotly zapewnia wiele sposobów, w jakie można nadawać style i dostosowywać wykres do własnych potrzeb po potwierdzeniu poprawności wyświetlanych na nim informacji. Wprowadzimy pewne zmiany w początkowym wywołaniu `px.bar()`, a następnie kolejne modyfikacje obiektu `fig` po jego utworzeniu.

Nadawanie stylu wykresowi rozpoczynamy od dodania tytułu i etykiet dla poszczególnych osi.

Plik `python_repos_visual.py`:

```
--cięcie--  
# Utworzenie wizualizacji.  
title = "Oznaczone największą liczbą gwiazdek projekty Pythona w serwisie GitHub"  
labels = {'x': 'Repozytorium', 'y': 'Gwiazdki'}  
fig = px.bar(x=repo_names, y=stars, title=title, labels=labels)  
  
fig.update_layout(title_font_size=28, xaxis_title_font_size=20, ❶  
                  yaxis_title_font_size=20)  
fig.show()
```

Na początku dodaliśmy tytuł i etykiety dla poszczególnych osi, podobnie jak to miało miejsce w rozdziałach 15. i 16. Następnie metoda `fig.update_layout()` została użyta do modyfikacji określonych elementów wykresu (patrz wiersz ❶). Plotly wykorzystuje konwencję, zgodnie z którą aspekty elementów wykresu są połączone znakami podkreślenia. Gdy lepiej poznasz dokumentację plotly, zaczniesz dostrzegać spójne wzorce w tym, jak różne elementy wykresów są nazywane i modyfikowane. W omawianym przykładzie wielkość czcionki tytułu wynosi 28, a czcionka dla osi ma wielkość 20. Na rysunku 17.2 pokazałem wykres po modyfikacjach.



Rysunek 17.2. Wykres po wprowadzonych modyfikacjach stylu

## Dodanie własnych podpowiedzi

W plotly umieszczenie kurSORA nad słupkiem wykresu spowoduje wyświetlenie informacji o tym, co przedstawia dany słupek. Wyświetlenie tego rodzaju informacji jest najczęściej nazywane *podpowiedzią*; w omawianym przykładzie podana będzie liczba gwiazdek zdobytych przez projekt. Przygotujemy teraz własną podpowiedź, aby wyświetlany był również opis danego projektu.

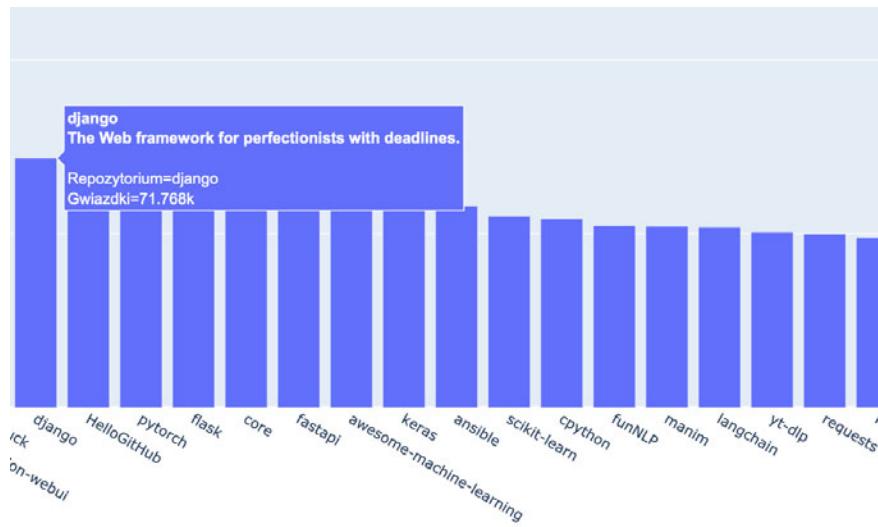
Konieczne jest pobranie danych dodatkowych w celu wygenerowania podpowiedzi.

Plik `python_repos_visual.py`:

```
--cięcie--  
# Przetworzenie informacji o repozytoriach.  
repo_dicts = response_dict['items']  
repo_names, stars, hover_texts = [], [], [] ❶  
for repo_dict in repo_dicts:  
    repo_names.append(repo_dict['name'])  
    stars.append(repo_dict['stargazers_count'])  
  
    # Przygotowanie podpowiedzi.  
    owner = repo_dict['owner']['login'] ❷  
    description = repo_dict['description']  
    hover_text = f'{owner}<br />{description}' ❸  
    hover_texts.append(hover_text)  
  
# Utworzenie wizualizacji.  
title = "Oznaczone największą liczbą gwiazdek projekty Pythona w serwisie GitHub"  
labels = {'x': 'Repozytorium', 'y': 'Gwiazdkii'}  
fig = px.bar(x=repo_names, y=stars, title=title, labels=labels, ❹  
              hover_name=hover_texts)  
  
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,  
                  yaxis_title_font_size=20)  
  
fig.show()
```

W wierszu ❶ definiujemy pustą listę o nazwie `hover_texts`, przeznaczoną do przechowywania tekstu, który będzie wyświetlany dla każdego projektu. W pętli przetwarzamy dane oraz pobieramy informacje o właścicielu i opis poszczególnych projektów (patrz wiersz ❷). Plotly pozwala używać kodu HTML w elementach tekstowych, więc dla etykiet generujemy ciągi tekstowe ze znakami nowego wiersza (`<br />`) między nazwą użytkownika projektu i jego opisem (patrz wiersz ❸). Wymienione etykiety są przechowywane na liście `hover_texts`.

W wywołaniu `px.bar()` dodajemy argument `hover_name` i przypisujemy mu utworzoną przed chwilą listę `hover_texts` (patrz wiersz ❹). Takie samo podejście zostało zastosowane podczas dostosowywania mapy pokazującej globalną aktywność sejsmiczną ziemi. W trakcie tworzenia poszczególnych słupków wykresu przez plotly etykiety zostaną pobrane z listy i wyświetcone tylko po umieszczeniu kurSORA myszy na słupku. Na rysunku 17.3 pokazałem wygenerowany wykres.



Rysunek 17.3. Umieszczenie kurSORA myszy na słupku powoduje wyświetlenie informacji o nazwie użytkownika projektu i jego opis

## Dodawanie łączy do wykresu

Ponieważ plotly pozwala stosować kod HTML w elementach tekstowych, bardzo łatwo można na słupkach zdefiniować łącza prowadzące do witryn internetowych. Użyjemy etykiet dla osi X jako sposobu pozwalającego użytkownikowi przejść na stronę projektu w serwisie GitHub. Konieczne jest pobranie adresu URL z danych i wykorzystanie go podczas generowania etykiet.

Plik `python_repos_visual.py`:

```
--cięcie--
# Przetworzenie informacji o repozytoriach.
repo_dicts = response_dict['items']
repo_links, stars, hover_texts = [], [], [] ❶
for repo_dict in repo_dicts:
    # Zamiana nazwy repozytorium na łącze.
    repo_name = repo_dict['name']
    repo_url = repo_dict['html_url'] ❷
    repo_link = f"<a href='{repo_url}'>{repo_name}</a>" ❸
    repo_links.append(repo_link)

    stars.append(repo_dict['stargazers_count'])
--cięcie--

# Utworzenie wizualizacji.
title = "Oznaczone największą liczbą gwiazdek projekty Pythona w serwisie GitHub"
labels = {'x': 'Repozytorium', 'y': 'Gwiazdki'}
fig = px.bar(x=repo_links, y=stars, title=title, labels=labels
              hover_name=hover_texts)
```

```
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,  
                  yaxis_title_font_size=20)  
  
fig.show()
```

---

Nazwę tworzonej listy zmieniamy z `repo_names` na `repo_links`, aby znacznie dokładniej przedstawić rodzaj informacji, które zostaną zamieszczone na wykresie (patrz wiersz ①). Następnie adres URL projektu jest pobierany z `repo_dict` i przypisywany zmiennej tymczasowej `repo_url` (patrz wiersz ②). W wierszu ③ generujemy łącze do projektu. Wykorzystujemy przy tym znacznik łącza w HTML, który ma postać `<a href='URL'>tekst łącza</a>`. Pozostało już tylko umieścić to łącze na liście `repo_links`.

W trakcie wywołania `px.bar()` używamy listy `repo_links` dla wartości X na wykresie. Wygenerowany wykres wygląda tak samo jak w poprzednim przykładzie. Jednak teraz możesz kliknąć dowolny słupek na wykresie, a zostaniesz przekierowany na stronę danego projektu w serwisie GitHub. W ten sposób przygotowaliśmy interaktywną, dostarczającą wielu użytecznych informacji wizualizację danych pobranych za pomocą API!

## Dostosowanie kolorów znaczników do własnych potrzeb

Gdy wykres został utworzony, praktycznie każdy jego element można dostosować do własnych potrzeb za pomocą metody uaktualnienia. Wcześniej pokazałem przykład użycia metody `update_layout()`. Inną metodą tego typu jest `update_traces()`, która pozwala dostosować do własnych potrzeb dane wyświetlane na ekranie.

Zmienimy teraz kolor słupków na ciemnoniebieski i dodamy przezroczystość.

---

```
--cięcie--  
fig.update_layout(title_font_size=28, xaxis_title_font_size=20,  
                  yaxis_title_font_size=20)  
  
fig.update_traces(marker_color='SteelBlue', marker_opacity=0.6)  
  
fig.show()
```

---

W plotly określenie *trace* odwołuje się do kolekcji danych na wykresie. Metoda `update_traces()` może pobrać wiele różnych argumentów. Każdy, który rozpoczyna się od `marker_`, wpływa na znaczniki znajdujące się na ekranie. W omawianym przykładzie znaczniki będą miały kolor 'SteelBlue', można użyć dowolnej nazwy koloru w CSS. Przejrzystość została zdefiniowana na 0.6. Wartość 1.0 oznacza kolor całkowicie pokrywający, natomiast wartość 0 oznacza pełną przezroczystość.

## Więcej o plotly i API GitHub

Dokumentacja plotly jest obszerna i doskonale zorganizowana. Jednak trudne może okazać się ustalenie, od czego rozpocząć jej przeglądanie. Dobrym początkiem będzie artykuł zatytułowany *Plotly Express in Python* (<https://plotly.com/python/plotly-express/>). Znajdziesz w nim ogólne omówienie wszystkich wykresów, jakie można przygotować za pomocą plotly express. Artykuł zawiera również łącza do dłuższych artykułów o poszczególnych typach wykresów.

Jeżeli chcesz dowiedzieć się, jak jeszcze lepiej dostosowywać do własnych potrzeb wykresy plotly, wiele cennych informacji na ten temat znajdziesz w artykule *Styling Plotly Express Figures in Python* (<https://plotly.com/python/styling-plotly-express/>). Zamieszczone w nim informacje pozwolą poszerzyć wiedzę zdobytą w rozdziałach 15., 16. i 17.

Więcej informacji na temat API GitHub znajdziesz w oficjalnej dokumentacji na stronie <https://docs.github.com/en/rest>. Dowiesz się z niej m.in., jak pobierać różne informacje z serwisu GitHub. Aby dowiedzieć się więcej niż podczas pracy nad tym projektem, skorzystaj z sekcji wyszukiwania na podanej stronie. Jeżeli masz konto w tym serwisie, możesz pracować z własnymi danymi, a także z publicznie dostępnymi danymi repozytoriów innych użytkowników.

## Hacker News API

Aby pokazać, jak można wykorzystać wywołania API w innych witrynach internetowych, zajrzymy do witryny Hacker News (<https://news.ycombinator.com/>). W witrynie tej użytkownicy udostępniają artykuły dotyczące programowania i technologii oraz zachęcają do prowadzenia żywych dyskusji na tematy poruszane w tych artykułach. API witryny Hacker News zapewnia dostęp do wszystkich danych związanych z przekazywanymi artykułami i komentarzami publikowanymi w witrynie. W celu skorzystania z tych możliwości nie jest wymagana żadna rejestracja.

Poniżej przedstawiłem wywołanie, które w trakcie powstawania tej książki zwracało informacje o najpopularniejszym artykule:

---

<https://hacker-news.firebaseio.com/v0/item/31353677.json>

---

Gdy ten adres URL wpiszesz w przeglądarce WWW, zobaczyś tekst wy wyświetlony w nawiasie klamrowym, co oznacza, że jest to słownik. Jednak samą odpowiedź trudno będzie przeanalizować bez uprzedniego sformatowania. Dlatego użyjemy tego adresu wraz z metodą `json.dumps()`, podobnie jak w poprzednim rozdziale podczas zapisywania danych o trzęsieniach ziemi. To pozwoli sprawdzić, jakie informacje otrzymujemy o wskazanym artykule.

## Plik hn\_article.py:

---

```
import requests
import json

# Wykonanie wywołania API i zachowanie otrzymanej odpowiedzi.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Kod stanu: {r.status_code}")

# Analiza struktury danych.
response_dict = r.json()
response_string = json.dumps(response_dict, indent=4)
print(response_string) ❶
```

---

Każde polecenie w tym programie powinno być Ci znane, ponieważ korzystaliśmy z nich w dwóch poprzednich rozdziałach. Podstawowa różnica jest taka, że w tym miejscu sformatowany ciąg tekstowy odpowiedzi jest wyświetlany (patrz wiersz ❶), a nie zapisywany do pliku, ponieważ te dane nie są zbyt obszerne.

Dane wyjściowe mają postać słownika informacji o artykule o identyfikatorze 31353677.

```
{
    "by": "sohkamyung",
    "descendants": 302, ❶
    "id": 31353677,
    "kids": [ ❷
        31354987,
        31354235,
        --cięcie--
    ],
    "score": 785,
    "time": 1652361401,
    "title": "Astronomers reveal first image of the black hole at the heart of our
    ↪galaxy", ❸
    "type": "story",
    "url": "https://public.nrao.edu/news/.../" ❹
}
```

---

Powyższy słownik zawiera wiele kluczy, które można wykorzystać podczas dalszej pracy. Klucz 'descendants' przechowuje liczbę komentarzy opublikowanych dla danego artykułu (patrz wiersz ❶). Z kolei klucz 'kids' dostarcza identyfikator wszystkich komentarzy opublikowanych bezpośrednio w odpowiedzi na dany artykuł (patrz wiersz ❷). Każdy z tych komentarzy również może zostać skomentowany, więc liczba komentarzy dla elementu 'descendants' jest zwykle większa niż dla elementu 'kids'. Dostępny jest też tytuł komentowanego artykułu (patrz wiersz ❸) oraz adres URL artykułu (patrz wiersz ❹).

Przedstawiony tutaj adres URL zwraca prostą listę wszystkich identyfikatorów aktualnie najpopularniejszych artykułów w Hacker News.

---

<https://hacker-news.firebaseio.com/v0/topstories.json>

---

Możemy go wykorzystać do ustalenia, które artykuły są obecnie wyświetlane na stronie głównej, a następnie wygenerować serię wywołań API podobnych do wcześniej przeanalizowanych. Takie podejście pozwala przeanalizować wszystkie artykuły wyświetcone na stronie głównej Hacker News.

*Plik hn\_submissions.py:*

```
from operator import itemgetter

import requests

# Wykonanie wywołania API i zachowanie otrzymanej odpowiedzi.
url = 'https://hacker-news.firebaseio.com/v0/topstories.json' ①
r = requests.get(url)
print(f"Kod stanu: {r.status_code}")

# Przetworzenie informacji o każdym artykule.
submission_ids = r.json() ②
submission_dicts = [] ③
for submission_id in submission_ids[:5]:
    # Przygotowanie oddzielnego wywołania API dla każdego artykułu.
    url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json" ④
    r = requests.get(url)
    print(f"id: {submission_id}\tstatus: {r.status_code}")
    response_dict = r.json()

    # Utworzenie słownika dla każdego artykułu.
    submission_dict = { ⑤
        'title': response_dict['title'],
        'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
        'comments': response_dict['descendants'],
    }
    submission_dicts.append(submission_dict) ⑥

submission_dicts = sorted(submission_dicts, key=itemgetter('comments'), ⑦
                         reverse=True)

for submission_dict in submission_dicts: ⑧
    print(f"\nTytuł artykułu: {submission_dict['title']}")
    print(f"Łącze do dyskusji: {submission_dict['hn_link']}")
    print(f"Liczba komentarzy: {submission_dict['comments']}")
```

---

Rozpoczynamy od przygotowania wywołania API oraz wyświetlenia kodu stanu otrzymanej odpowiedzi (patrz wiersz ①). Wynikiem wykonania tego wywołania API jest lista zawierająca identyfikatory 500 najpopularniejszych artykułów

w witrynie Hacker News w chwili wykonania tego zapytania. Następnym krokiem jest konwersja tekstu odpowiedzi na listę Pythona (patrz wiersz ②), którą przechowujemy w zmiennej `submission_ids`. Te identyfikatory zostaną użyte do przygotowania słowników przechowujących informacje o poszczególnych artykułach.

W wierszu ③ przygotowujemy pustą listę o nazwie `submission_dicts`, przeznaczoną do przechowywania wspomnianych słowników. Następnie przeprowadzamy iterację przez 5 pierwszych identyfikatorów. Dla każdego z nich przygotowujemy nowe wywołanie API przez wygenerowanie adresu URL zawierającego wartość bieżącą zmiennej `submission_id` (patrz wiersz ④). Stan wykonania każdego żądania zostaje wyświetlony, aby można było sprawdzić, czy wywołanie zakończyło się sukcesem.

W wierszu ⑤ tworzymy słownik dla każdego aktualnie przetwarzanego artykułu. W tym słowniku umieszczamy tytuł artykułu oraz łącze prowadzące do strony, na której toczy się dyskusja dotycząca danego artykułu. Następnie każdy słownik `submission_dict` umieszczamy na liście `submission_dicts` (patrz wiersz ⑥).

Artykuły w witrynie Hacker News są układane w kolejności uwzględniającej liczbę zdobytych punktów, które są przyznawane za wiele różnych rzeczy, takich jak na przykład liczba ocen danego artykułu, liczba opublikowanych do niego komentarzy oraz data publikacji artykułu. Listę przygotowanych słowników chcemy posortować według liczby komentarzy. W tym celu używamy funkcji o nazwie `itemgetter()`, która pochodzi z modułu `operator` (patrz wiersz ⑦). Do tej funkcji przekazujemy klucz 'comments' i pobieramy przypisaną mu wartość z każdego słownika znajdującego się na liście. Następnie funkcja `sorted()` wykorzystuje tę wartość jako podstawę podczas sortowania listy. Listę sortujemy w kolejności malejącej, więc najczęściej komentowane artykuły będą umieszczone na początku.

Po posortowaniu listy przeprowadzamy przez nią iterację (patrz wiersz ⑧) i wyświetlamy trzy rodzaje informacji o poszczególnych najpopularniejszych artykułach: tytuł artykułu, łącze do strony, na której toczy się dyskusja dotycząca danego artykułu, oraz aktualną liczbę komentarzy opublikowanych do danego artykułu:

---

```
Kod stanu: 200
id: 31390506    status: 200
id: 31389893    status: 200
id: 31390742    status: 200
--cięcie--
```

Tytuł artykułu: Fly.io: The reclaimer of Heroku's magic  
Łącze do dyskusji: <http://news.ycombinator.com/item?id=31390506>  
Liczba komentarzy: 134

Tytuł artykułu: The weird Hewlett Packard FreeDOS option  
Łącze do dyskusji: <http://news.ycombinator.com/item?id=31389893>  
Liczba komentarzy: 64

Tytuł artykułu: Modern JavaScript Tutorial

Łącze do dyskusji: <http://news.ycombinator.com/item?id=31390742>

Liczba komentarzy: 20

--cięcie--

---

Podobny proces możesz zastosować, aby uzyskać dostęp do informacji pobieranych za pomocą innych API i je przeanalizować. Mając przygotowane dane, możesz utworzyć wizualizacje pokazujące, które artykuły najbardziej zainspirowały użytkowników do prowadzenia aktywnej dyskusji. Są to również podstawy dla aplikacji umożliwiającej użytkownikom dostosowanie wyglądu witryn takich jak Hacker News. Aby dowiedzieć się więcej na temat informacji, do których dostęp można uzyskać za pomocą API Hacker News, zajrzyj do dokumentacji na stronie <https://github.com/HackerNews/API/>.

**UWAGA** *Hacker News czasami pozwala wspierającym ten serwis firmom publikować specjalne posty rekrutacyjne. Dla takich postów komentarze są wyłączone. Jeżeli omówiony program uruchomisz, gdy jeden z takich postów zostanie opublikowany, to otrzymasz błąd typu KeyError. Wówczas kod tworzący słownik submission\_dict możesz umieścić w konstrukcji try-except, aby pomijać te posty.*

## ZRÓB TO SAM

**17.1. Inne języki programowania.** Zmodyfikuj wywołanie API w programie `python_repos.py`, aby generowało wykres pokazujący najpopularniejsze projekty także w innych językach programowania. Spróbuj przygotować wykres dla języków takich jak *JavaScript, Ruby, C, Java, Perl, Haskell i Go*.

**17.2. Aktywne dyskusje.** Wykorzystując dane zebrane w programie `hn_submissions.py`, utwórz wykres słupkowy pokazujący najaktywniejsze dyskusje obecnie prowadzone w witrynie Hacker News. Wysokość słupka powinna odpowiadać liczbie komentarzy opublikowanych dla danego artykułu. Z kolei etykieta dla każdego słupka powinna zawierać tytuł artykułu. Ponadto każdy słupek powinien być łączem prowadzącym na stronę dyskusji dotyczącej danego artykułu. Jeżeli podczas tworzenia wykresu otrzymasz błąd KeyError, użyj bloku try-except, aby pominąć posty reklamowe.

**17.3. Testowanie programu `python_repos.py`.** W programie `python_repos.py` wyświetliśmy wartość zmiennej `status_code`, aby mieć pewność, że wywołanie API zakończyło się powodzeniem. Utwórz program o nazwie `test_python_repos.py`, który wykorzysta moduł `pytest` do przeprowadzenia asercji, czy wartość `status_code` wynosi 200. Spróbuj znaleźć jeszcze inne asercje, które można zastosować, na przykład czy liczba zwróconych elementów jest zgodna z oczekiwaniemi, a całkowita liczba repozytoriów jest większa od pewnej zdefiniowanej wartości.

**17.4. Dalsze wyjaśnienie.** Zapoznaj się z dokumentacją plotly oraz API GitHub lub API Hacker News. Zdobyte w ten sposób informacje wykorzystaj, aby sprawdzić, czy jesteś w stanie dostosować do własnych potrzeb style wykresów wygenerowanych w tym rozdziale, lub aby pobrać inne dane, dla których później utworzysz wizualizacje. Jeżeli chcesz poznać jeszcze inne API, zajrzyj do wspomnianego w tym rozdziale repozytorium GitHub dotyczącego API (<https://github.com/public-apis/public-apis>).

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak użyć API do utworzenia samodzielnych programów, które automatycznie będą pobierać niezbędne im dane, a następnie wykorzystywać je do przygotowania wizualizacji. Skorzystaliśmy z API serwisu GitHub, aby poznać najczęściej oznaczane gwiazdką projekty Pythona zamieszczone w GitHub, oraz pokróćce spojrzaliśmy na API witryny Hacker News. Zobaczyłeś, jak można zastosować pakiet `requests` do automatycznego wykonywania wywołań API do serwisu GitHub, a także jak przetwarzać wyniki tych wywołań. Poznałeś także wybrane ustawienia `plotly`, które jeszcze bardziej pomagają dostosować do własnych potrzeb wygląd generowanych wykresów.

W ostatnim projekcie w tej książce wykorzystamy framework Django do zbudowania aplikacji internetowej.

# 18

## Rozpoczęcie pracy z Django



WRAZ Z ROZWOJEM INTERNETU ZACIERA SIĘ GRANICA MIĘDZY WITRYNAMI INTERNETOWYMI I APLIKACJAMI MOBILNYMI. WITRYNY I APLIKACJE POMAGAJĄ UŻYTKOWNIKOM PRACOWAĆ Z DANYMI NA RÓŻNE SPOSÓBY.

Na szczęście można wykorzystać Django do opracowania pojedynczego projektu, który będzie działał w charakterze zarówno dynamicznej witryny internetowej, jak i zestawu aplikacji mobilnych. *Django* to najpopularniejszy w ekosystemie Pythona *framework internetowy* — zestaw narzędzi pomocnych w tworzeniu interaktywnych aplikacji internetowych. W tym rozdziale dowiesz się, jak wykorzystać Django do opracowania projektu o nazwie *Learning Log*. Zbudujemy po prostu system dziennika internetowego pozwalającego na monitorowanie tego, czego dowiedziałeś się na określone tematy.

Zaczniemy od opracowania specyfikacji dla projektu, a następnie zdefiniujemy modele dla danych, z którymi będzie działała aplikacja. Witrynę administracyjną Django wykorzystamy do wstawienia pewnych danych początkowych, a następnie przejdziemy do tworzenia widoków i szablonów, za pomocą których Django będzie budować strony naszej witryny internetowej.

Django może reagować na żądania konkretnych stron, a także pomagać odczytywać i zapisywać informacje w bazie danych, zarządzać użytkownikami itd. W rozdziałach 19. i 20. dopracujemy projekt *Learning Log* oraz wdrożymy go na serwerze, abyś mógł (wraz ze swoimi przyjaciółmi) z niego korzystać.

# Przygotowanie projektu

Rozpoczynając pracę nad projektem, najpierw trzeba go opisać za pomocą specyfikacji, określonej również mianem *spec*. Po jasnym sprecyzowaniu celów można przystąpić do określenia zadań, których wykonanie pozwoli osiągnąć założone cele.

W tym podrozdziale zajmiemy się opracowaniem specyfikacji dla Learning Log oraz rozpoczęmy pracę nad pierwszą fazą projektu. Obejmuje ona przygotowanie środowiska wirtualnego, w którym zostanie zbudowany ten projekt, oraz utworzenie początkowych aspektów projektu Django.

## Opracowanie specyfikacji

Pełna i szczegółowa specyfikacja projektu zawiera informacje o jego celach, opis funkcjonalności, a także omówienie interfejsu użytkownika. Podobnie jak to jest w przypadku każdego innego dobrego projektu lub biznesplanu, specyfikacja powinna pomóc programistom zachować koncentrację na projekcie i ułatwić im jego realizację. W tym miejscu nie przygotujemy pełnej specyfikacji projektu, ale wyraźnie wskażemy kilka celów, na których osiągnięciu będziemy się koncentrować. Oto specyfikacja budowanego projektu:

Utworzmy aplikację internetową o nazwie *Learning Log*, pozwalającą użytkownikom zapisywać interesujące ich informacje i tym samym dodawać wpisy do dziennika podczas poznawania danego zagadnienia. Strona główna aplikacji *Learning Log* powinna zawierać opis wyjaśniający przeznaczenie tej witryny internetowej oraz zachęcać użytkowników do zarejestrowania się lub zalogowania. Po zalogowaniu użytkownik powinien mieć możliwość utworzenia nowego tematu, dodania nowego wpisu, a także odczytania i edytowania istniejących wpisów.

Kiedy poznajesz nowe zagadnienie, prowadzenie dziennika, w którym zapisujesz to, czego się nauczyłeś, może być pomocne w monitorowaniu postępów oraz podczas ponownego przeglądania zebranych informacji. Dzięki dobrej aplikacji ten proces może być znacznie efektywniejszy.

## Utworzenie środowiska wirtualnego

Aby pracować z Django, trzeba zacząć od skonfigurowania środowiska wirtualnego. Wspomniane *środowisko wirtualne* to miejsce w systemie, w którym można instalować pakiety i odizolować je od wszystkich pozostałych pakietów Pythona. Oddzielenie bibliotek poszczególnych projektów jest korzystne i będzie niezbędne podczas wdrażania aplikacji *Learning Log* na serwerze WWW, czym się zajmiemy w rozdziale 20.

Utwórz nowy katalog dla projektu i nadaj mu nazwę *learning\_log*. W powłoce przejdź do nowo utworzonego katalogu, a następnie utwórz w nim środowisko wirtualne. Do utworzenia środowiska wirtualnego powinieneś wykorzystać wymienione poniżej polecenie:

---

```
learning_log$ python -m venv ll_env
learning_log$
```

---

W powyższym poleceniu korzystamy z modułu `venv` i używamy go do utworzenia środowiska wirtualnego o nazwie `ll_env` (zwróć uwagę na umieszczenie w nazwie dwóch małych liter `l`). Jeżeli do uruchamiania programów lub Pythona w sesji powłoki używasz innego polecenia niż `python`, np. `python3`, musisz odpowiednio zmodyfikować przedstawione wcześniej polecenie.

## Aktywacja środowiska wirtualnego

Skoro mamy już skonfigurowane środowisko wirtualne, następnym krokiem jest jego aktywacja za pomocą poniższego polecenia:

---

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$
```

---

To polecenie powoduje wykonanie skryptu `activate` znajdującego się w `ll_env/bin`. Kiedy środowisko wirtualne jest aktywne, jego nazwa będzie wyświetiana w nawiasie, przed znakiem zachęty powłoki. Teraz możesz instalować pakiety w tym środowisku oraz korzystać z tych już zainstalowanych. Pakiety instalowane w `ll_env` będą dostępne tylko wtedy, gdy zawierające je środowisko będzie aktywne.

**UWAGA** W systemie Windows do aktywacji środowiska wirtualnego należy użyć polecenia `ll_env\Scripts\activate` (bez słowa `source` na początku). Jeśli używasz wiersza poleceń PowerShell, słowo `Activate` musi zaczynać się wielką literą.

Aby zakończyć korzystanie ze środowiska wirtualnego, wystarczy wydać polecenie `deactivate`:

---

```
(ll_env)learning_log$ deactivate
learning_log$
```

---

Warto również dodać, że środowisko wirtualne stanie się nieaktywne, kiedy zamknieniemy powłokę, w której działało.

## Instalacja frameworka Django

Po utworzeniu i aktywowaniu środowiska wirtualnego możemy przystąpić do instalacji frameworka Django:

---

```
(ll_env)learning_log$ pip install --upgrade pip
(ll_env)learning_log$ pip install django
Collecting django
```

```
--cięcie--  
Installing collected packages: sqlparse, asgiref, django  
Successfully installed asgiref-3.5.2 django-4.1 sqlparse-0.4.2  
(11_env)learning_log$
```

---

Narzędzie pip pobiera zasoby z wielu źródeł, więc jest uaktualniane dość często. Dobrym rozwiążaniem jest uaktualnienie narzędzia pip po utworzeniu nowego środowiska wirtualnego.

Ponieważ działamy w środowisku wirtualnym, polecenie przeznaczone do instalacji Django jest takie samo dla wszystkich systemów operacyjnych. Nie ma konieczności użycia opcji `--user`, nie trzeba również wydawać dłuższych poleceń, takich jak `python -m pip install nazwa_pakietu`. Warto pamiętać, że framework Django będzie dostępny tylko wtedy, gdy zawierające go środowisko wirtualne będzie aktywne.

**UWAGA** *Nowe wydania frameworka Django pojawiają się mniej więcej co 8 miesięcy. Istnieje więc duże prawdopodobieństwo, że będziesz mieć zainstalowaną nowszą wersję niż wymieniona nieco wcześniej w tym rozdziale. Ten projekt powinien działać również w nowszych wersjach Django. Jeżeli chcesz mieć pewność, że używasz tej samej wersji, z której korzystałem podczas tworzenia projektu, wydaj polecenie `pip install django==4.1.*`. Spowoduje to instalację najnowszego dostępnego wydania Django 4.1. Jeżeli masz jakiekolwiek problemy związane z używaną wersją Django, zajrzyj na stronę zawierającą materiały przygotowane dla tej książki, pod adresem [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/).*

## Utworzenie projektu w Django

Bez opuszczania aktywnego środowiska wirtualnego (pamiętaj, aby sprawdzić, czy wyświetlane jest `ll_env` w nawiasie) wydaj poniższe polecenie, które spowoduje utworzenie nowego projektu:

```
(11_env)learning_log$ django-admin startproject ll_project . ❶  
(11_env)learning_log$ ls ❷  
ll_env ll_project manage.py  
(11_env)learning_log$ ls ll_project ❸  
__init__.py settings.py urls.py wsgi.py
```

---

Polecenie `startproject` w wierszu ❶ nakazuje Django utworzyć nowy projekt o nazwie `ll_project`. Kropka na końcu polecenia powoduje utworzenie nowego projektu wraz ze strukturą katalogów ułatwiającą wdrożenie aplikacji na serwerze, gdy zakończymy już nad nią pracę.

**UWAGA** *Na zapomnij o wspomnianej kropce, ponieważ w przeciwnym razie możesz napotkać pewne problemy z konfiguracją podczas wdrażania aplikacji. Gdy zapomnisz kropki, usuń utworzone pliki i katalog projektu (poza oczywiście `ll_env`) i ponownie wydaj powyższe polecenie.*

Wynik wykonania polecenia `ls` (dir w systemie Windows) pokazuje (patrz wiersz ②), że framework Django utworzył nowy katalog o nazwie `ll_project`. Utworzony został również plik o nazwie `manage.py`, który jest niewielkim programem przeznaczonym do wydawania poleceń odpowiednim komponentom frameworka Django. Polecenia tego programu wykorzystamy do zarządzania zadaniami takimi jak praca z bazami danych i uruchamianie serwerów.

Katalog `ll_project` zawiera cztery pliki (patrz wiersz ③), z których najważniejsze to: `settings.py`, `urls.py` i `wsgi.py`. Plik `settings.py` kontroluje sposób współdziałania Django z systemem operacyjnym oraz pozwala na zarządzanie projektem. Zmodyfikujemy teraz wybrane ustawienia, a później, wraz z rozwojem projektu, dodamy jeszcze kilka nowych. Plik `urls.py` wskazuje Django, które strony powinny zostać utworzone w odpowiedzi na żądania pochodzące z przeglądarki internetowej. Z kolei plik `wsgi.py` pomaga udostępniać pliki utworzone przez Django. Nazwa `wsgi.py` to skrót od *web server gateway interface*.

## Utworzenie bazy danych

Ponieważ większość informacji związanych z projektem jest przez Django przechowywana w bazie danych, konieczne jest utworzenie bazy danych, z którą framework będzie mógł pracować. Aby utworzyć bazę danych dla projektu *Learning Log*, wydaj poniższe polecenie (nadal w aktywnym środowisku wirtualnym):

---

```
(11_env)learning_log$ python manage.py migrate
Operations to perform: ①
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  --cięcie--
  Applying sessions.0001_initial... OK
(11_env)learning_log$ ls ②
db.sqlite3  11_env  ll_project  manage.py
```

---

Za każdym razem, gdy modyfikujemy bazę danych, mówimy o przeprowadzeniu *migracji*. Wydanie polecenia `migrate` po raz pierwszy nakazuje Django sprawdzić, czy baza danych odpowiada bieżącemu stanowi projektu. Kiedy po raz pierwszy wydasz to polecenie w nowym projekcie wykorzystującym SQLite (więcej o SQLite dowiesz się już za chwilę), Django utworzy nową bazę danych. Z wiersza ① dowiadujemy się, że przygotowuje bazę danych do przechowywania informacji niezbędnych do przeprowadzania zadań administracyjnych oraz związanych z uwierzytelnianiem użytkowników.

Wykonanie polecenia `ls` ujawnia, że Django utworzył plik o nazwie `db.sqlite3` (patrz wiersz ②). SQLite to baza danych oparta na pojedynczym pliku. Doskonale sprawdzi się podczas tworzenia prostych aplikacji, ponieważ korzystając z niej, nie trzeba zwracać zbyt dużej uwagi na kwestie związane z zarządzaniem bazą danych.

**UWAGA** W aktywnym środowisku wirtualnym należy do wydawania poleceń `manage.py` używać polecenia `python`, nawet jeśli standardowo uruchamiasz programy za pomocą innego, np. `python3`. W środowisku wirtualnym polecenie `python` odwołuje się do wersji Pythona użytej do utworzenia tego środowiska wirtualnego.

## Przegląd projektu

Upewnijmy się teraz o prawidłowym przygotowaniu projektu przez Django. Wydaj polecenie `runserver`, tak jak pokazałem poniżej:

```
(11_env)learning_log$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

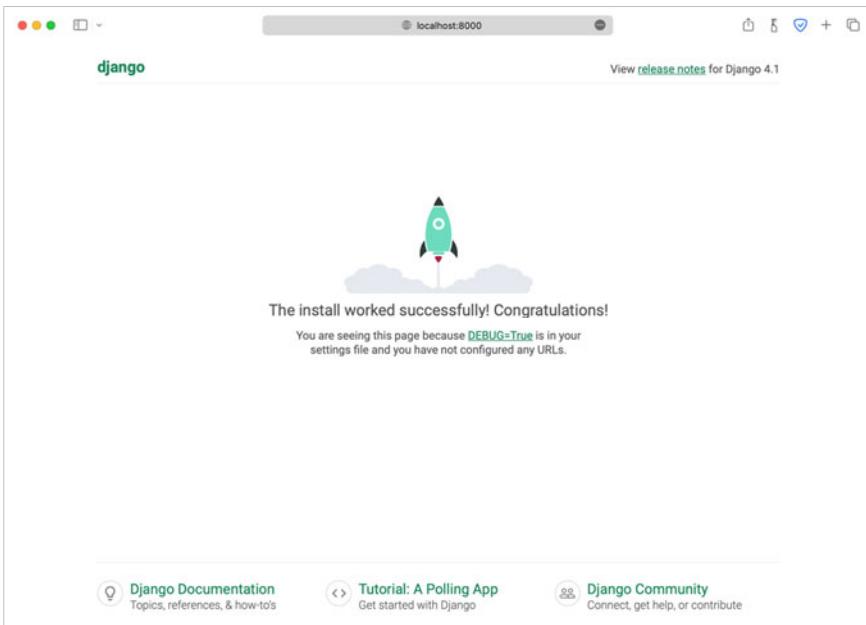
System check identified no issues (0 silenced). ❶
May 19, 2022 - 21:52:35
Django version 4.1, using settings '11_project.settings' ❷
Starting development server at http://127.0.0.1:8000/ ❸
Quit the server with CONTROL-C.
```

Django uruchamia serwer określany mianem *serwera programistycznego*, co pozwoli z kolei uruchomić projekt w systemie i sprawdzić, jak działa. Kiedy żądasz strony i podasz jej adres URL w przeglądarce internetowej, serwer Django odpowie na to żądanie, tworząc odpowiednią stronę i przekazując ją do przeglądarki internetowej.

W wierszu ❶ Django sprawdza, czy projekt został prawidłowo przygotowany. Następnie w wierszu ❷ podawana jest używana wersja frameworka Django oraz nazwa pliku ustawień. Z kolei w wierszu ❸ znajduje się adres URL, pod którym jest dostępny projekt. Adres URL w postaci `http://127.0.0.1:8000/` wskazuje, że projekt nasłuchuje żądań na porcie 8000 w komputerze lokalnym, nazywanym również `localhost`. Nazwa `localhost` odwołuje się do serwera, który przetwarza żądania jedynie w komputerze lokalnym. Nikt inny nie będzie miał dostępu do przygotowanych przez Ciebie stron internetowych.

Teraz otwórz przeglądarkę internetową i przejdź pod adres URL `http://localhost:8000/`, lub też `http://127.0.0.1:8000/`, jeśli pierwszy z wymienionych nie działa. Powinieneś zobaczyć wyświetlzoną stronę podobną do pokazanej na rysunku 18.1. Tę stronę framework Django tworzy, aby poinformować, że jak dotąd wszystko przebiega pomyślnie. Na razie pozostaw uruchomiony serwer. Jeżeli będziesz chciał go zatrzymać, wystarczy nacisnąć klawisze `Ctrl+C` w powloce, w której zostało wydane polecenie `runserver`.

**UWAGA** Jeżeli otrzymasz komunikat błędu, że port 8000 jest już używany przez inny proces, nakaż Django nasłuchiwanie na innym porcie. W tym celu wydaj polecenie `python manage.py runserver 8001`. Możesz podawać kolejne numery portów, dopóki nie znajdziesz otwartego portu.



Rysunek 18.1. Jak dotąd wszystko działa doskonale

## ZRÓB TO SAM

**18.1. Nowe projekty.** Aby jeszcze lepiej poznać możliwości oferowane przez framework Django, utwórz kilka pustych projektów i zobacz, z czego się składają. Utwórz nowy katalog o prostej nazwie, takiej jak `tik_gram` lub `insta_tok` (na zewnątrz katalogu `learning_log`), przejdź do nowego katalogu w powłoce i utwórz środowisko wirtualne. Zainstaluj Django i wydaj polecenie `django-admin.py startproject tg_project`. (upewnij się, że umieściłeś kropkę na końcu polecenia).

Zajrzyj do plików i katalogów utworzonych przez powyższe polecenie i porównaj je z zawartością projektu *Learning Log*. Procedurę powtórz kilkukrotnie, aż będziesz wiedzieć, co Django tworzy w nowym projekcie. Następnie usuń katalogi niepotrzebnych Ci projektów.

## Uruchomienie aplikacji

Na projekt Django składa się grupa poszczególnych *aplikacji*, które współpracują ze sobą, co pozwala na działanie projektu jako całości. W tym rozdziale utworzymy tylko jedną aplikację przeznaczoną do wykonywania większości zadań budowanego projektu. Natomiast w rozdziale 19. dodamy następną aplikację, odpowiedzialną za zarządzanie kontami użytkowników.

W otworzonym wcześniej oknie powłoki nadal powinieneś mieć uruchomiony serwer (polecamie runserver). Przejdź teraz do nowego okna powłoki (lub nowej karty w oknie terminala), a następnie do katalogu zawierającego program *manage.py*. Aktywuj środowisko wirtualne i wydaj polecenie startapp:

---

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$ python manage.py startapp learning_logs
(ll_env)learning_log$ ls ①
db.sqlite3 learning_logs ll_env ll_project manage.py
(ll_env)learning_log$ ls learning_logs/ ②
__init__.py admin.py apps.py migrations models.py tests.py views.py
```

---

Polecenie startapp *nazwa\_aplikacji* nakazuje Django utworzyć infrastrukturę niezbędną do zbudowania aplikacji. Jeśli spojrzysz teraz do katalogu projektu, zobaczysz nowy podkatalog o nazwie *learning\_logs* (patrz wiersz ①). Otwórz go i sprawdź, jakie pliki zostały wygenerowane przez Django (patrz wiersz ②). Najważniejszymi plikami są: *models.py*, *admin.py* i *views.py*. Pliku *models.py* będziemy używać do zdefiniowania danych, którymi będziemy chcieli zarządzać w tej aplikacji. Natomiast pozostałymi plikami (*admin.py* i *views.py*) zajmiemy się nieco później.

## Definiowanie modeli

Zastanów się przez chwilę nad danymi przeznaczonymi do użycia w aplikacji. Każdy użytkownik musi mieć możliwość utworzenia wielu różnych tematów podczas procesu nauki. Poszczególne wpisy będą powiązane z tematem i zostaną wyświetcone w postaci tekstu. Konieczne jest również przechowywanie znacznika czasu dla każdego wpisu, aby móc pokazać użytkownikowi, kiedy dany wpis został dokonany.

Otwórz plik *models.py* i spójrz na jego istniejącą zawartość.

*Plik models.py:*

---

```
from django.db import models

# Miejsce na Twoje modele.
```

---

Moduł o nazwie *models* został zimportowany automatycznie, a umieszczony w pliku komentarz zachęca do utworzenia własnych modeli. *Model* w Django wskazuje sposób pracy z danymi, które będą przechowywane w aplikacji. Model jest po prostu klasą, zawiera atrybuty i metody podobnie jak wszystkie omawiane wcześniej klasy. Poniżej przedstawiłem model przeznaczony do obsługi tematów, które będą przechowywane przez użytkowników:

---

```
from django.db import models

class Topic(models.Model):
    """Temat poznawany przez użytkownika."""
    text = models.CharField(max_length=200) ①
    date_added = models.DateTimeField(auto_now_add=True) ②

    def __str__(self): ③
        """Zwraca reprezentację modelu w postaci ciągu tekstowego."""
        return self.text
```

---

Utworzyliśmy klasę o nazwie `Topic` dziedziczącą po klasie `Model`, czyli klasie nadrzędej oferowanej przez Django, i definiującą podstawową funkcjonalność modelu. W nowej klasie `Topic` mamy tylko dwa atrybuty: `text` i `date_added`.

Atrybut `text` jest typu `CharField`, co oznacza dane składające się ze znaków, czyli po prostu tekst (patrz wiersz ①). Tego rodzaju danych można używać, gdy zachodzi potrzeba przechowywania niewielkiej ilości tekstu, na przykład imienia, tytułu lub nazwy miejscowości. Podczas definiowania atrybutu typu `CharField` konieczne jest poinformowanie Django, jaka ilość miejsca na te informacje powinna być zarezerwowana w bazie danych. W omawianym przykładzie podajemy wielkość 200 znaków (`max_length=200`), która powinna być wystarczająca dla praktycznie wszystkich tytułów.

Atrybut `date_added` jest typu `DateTimeField` — tego rodzaju dane przechowują informacje o dacie i godzinie (patrz wiersz ②). Przekazujemy argument `auto_add_now=True`, który nakazuje Django automatyczne przypisywanie temu atrybutowi bieżącej daty i godziny w chwili tworzenia nowego tematu przez użytkownika.

Dobrym pomysłem jest wskazanie Django, który atrybut ma być używany domyślnie podczas wyświetlania informacji dotyczących tematu. Jeżeli model ma metodę `__str__()`, Django wywołuje ją za każdym razem, gdy zachodzi potrzeba wygenerowania danych wyjściowych odwołujących się do egzemplarza tego modelu. W omawianym przykładzie wykorzystaliśmy metodę `__str__()`, która zwraca串tekstowy przechowywany w atrybucie `text` (patrz wiersz ③).

Jeżeli chcesz poznać różne rodzaje danych, jakie można stosować w modelach, zajrzyj do dokumentu zatytułowanego *Model Field Reference*, który znajdziesz na stronie <https://docs.djangoproject.com/en/4.1/ref/models/fields/>. Nie potrzebujesz wszystkich wymienionych w nich informacji, ale okażą się one użyteczne podczas opracowywania własnych aplikacji.

## Aktywacja modeli

Aby użyć modeli, trzeba frameworkowi Django nakazać dołączenie budowanej aplikacji do ogólnego projektu. Otwórz plik `settings.py` (katalogu `ll_project`) i odszukaj sekcję wskazującą aplikacje zainstalowane w projekcie.

## Plik settings.py:

---

```
--cięcie--  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]  
--cięcie--
```

---

Dodaj budowaną aplikację do tej listy, modyfikując `INSTALLED_APPS` w przedstawiony poniżej sposób:

```
--cięcie--  
INSTALLED_APPS = [  
    # Moje aplikacje.  
    'learning_logs',  
  
    # Domyślne aplikacje Django.  
    'django.contrib.admin',  
    --cięcie--  
]  
--cięcie--
```

---

Grupowanie aplikacji w projekcie ułatwia ich monitorowanie wraz z rozwojem projektu, gdy zaczyna on zawierać coraz więcej aplikacji. W powyższym fragmencie kodu możesz zobaczyć, że utworzyliśmy sekcję zatytułowaną *Moje aplikacje*, w której na razie znajduje się tylko jedna aplikacja — `learning_logs`. Bardzo duże znaczenie ma umieszczanie własnych aplikacji przed domyślnymi na wypadek, gdyby zachodziła konieczność nadpisywania zachowania aplikacji domyślnych zachowaniem zdefiniowanym w tworzonych aplikacjach.

Kolejnym krokiem jest nakazanie Django przeprowadzenia modyfikacji bazy danych tak, aby można było przechowywać w niej informacje powiązane z modelem `Topic`. Przejdź więc do powłoki, a następnie wydaj poniższe polecenie:

---

```
(11_env)learning_log$ python manage.py makemigrations learning_logs  
Migrations for 'learning_logs':  
    learning_logs/migrations/0001_initial.py:  
        - Create model Topic  
(11_env)learning_log$
```

---

Polecenie `makemigrations` nakazuje frameworkowi Django ustalenie sposobu modyfikacji bazy danych, aby można było w niej przechowywać informacje powiązane ze wszystkimi modelami zdefiniowanymi w aplikacji. Wyświetlone dane

wyjściowe pokazują, że framework wygenerował plik migracji o nazwie `0001_initial.py`. Ta migracja spowoduje utworzenie w bazie danych tabeli przeznaczonej dla modelu Topic.

Teraz zastosujemy tę migrację, co oznacza faktyczne wprowadzenie przez Django zmian w bazie danych:

```
(11_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
  Applying learning_logs.0001_initial... OK
```

Większość wyświetlonych danych wyjściowych jest identyczna z otrzymanymi po pierwszym wykonaniu polecenia `migrate`. Trzeba zwrócić szczególną uwagę na ostatni wiersz, w którym znajdują się informacje o wyniku przeprowadzonej migracji. Jeśli wszystko przebiegło prawidłowo i migracja w aplikacji `learning_logs` została zastosowana bez błędów, otrzymasz komunikat OK, jak w omawianym przykładzie.

Jeśli kiedykolwiek wystąpi potrzeba modyfikacji danych, którymi zarządza budowana tutaj aplikacja, będziesz musiał wykonać trzy pokazane wcześniej kroki: zmodyfikować plik `models.py`, wywołać `makemigrations` w aplikacji `learning_logs` i nakazać frameworkowi Django faktyczne przeprowadzenie migracji (polecenie `migrate`).

## Witryna administracyjna Django

Kiedy definiujesz modele dla aplikacji, Django może niezwykle ułatwić pracę z tymi modelami, jeśli skorzystasz z oferowanej przez framework tak zwanej *witryny administracyjnej*, która jest przeznaczona dla administratorów witryny, a nie jej zwykłych użytkowników. W tej sekcji skonfigurujemy witrynę administracyjną oraz użyjemy jej, aby dodać pewne tematy za pomocą modelu Topic.

### Konfiguracja superużytkownika

Django pozwala na utworzenie użytkownika, który będzie miał wszystkie uprawnienia w danej witrynie internetowej. Taki użytkownik jest nazywany *superużytkownikiem*. Wspomniane uprawnienia kontrolują działania, jakie mogą być podejmowane przez użytkownika. W przypadku najbardziej restrykcyjnych uprawnień użytkownik będzie mógł jedynie czytać informacje publicznie udostępnione w witrynie internetowej. Zarejestrowani użytkownicy zwykle mają możliwość czytania także prywatnych danych innych użytkowników oraz mają dostęp do informacji przeznaczonych wyłącznie dla członków danej społeczności. Aby móc efektywnie administrować aplikacją internetową, właściciel witryny zwykle potrzebuje mieć dostęp do wszystkich informacji przechowywanych w danej witrynie internetowej. Dobry administrator zachowuje szczególną ostrożność w przypadku

informacji wrażliwych użytkowników, ponieważ darzą oni dużym zaufaniem aplikacje, których używają.

Aby utworzyć superużytkownika w Django, wydaj poniższe polecenie i udzielaj odpowiedzi na wyświetlane pytania:

---

```
(11_env)learning_log$ python manage.py createsuperuser
Username (leave blank to use 'eric'): 11_admin ❶
Email address: ❷
Password:
Password (again): ❸
Superuser created successfully.
(11_env)learning_log$
```

---

Kiedy wydasz polecenie `createsuperuser`, Django poprosi o podanie nazwy użytkownika (patrz wiersz ❶). W omawianym przykładzie podałem `11_admin`, ale w tym miejscu możesz podać dowolnie wybraną nazwę użytkownika. Następnie możesz podać adres e-mail lub pozostawić to pole puste (patrz wiersz ❷). Hasło trzeba będzie podać dwukrotnie (patrz wiersz ❸).

**UWAGA** Pewne informacje wrażliwe mogą być ukryte w witrynie administracyjnej. Przykładowo Django tak naprawdę nie przechowuje wprowadzonego hasła, ale wygenerowany na jego podstawie ciąg tekstowy nazywany wartością hash. Za każdym razem, gdy wprowadzisz hasło, Django wygeneruje dla niego wartość hash i porówna ją z przechowywaną. Jeżeli obie wartości hash są identyczne, oznacza to, że użytkownik podał prawidłowe hasło i został uwierzytelniony. Porównywanie wartości hash jest bezpiecznym rozwiązaniem, ponieważ jeśli atakujący uzyska dostęp do bazy danych witryny internetowej, będzie w stanie odczytać jedynie przechowywane tam wartości hash, a nie hasła. Jeżeli witryna internetowa została prawidłowo skonfigurowana, jest praktycznie niemożliwe wygenerowanie hasel na podstawie wartości hash.

## Rejestracja modelu w witrynie administracyjnej

W witrynie administracyjnej Django pewne modele zostały umieszczone automatycznie, na przykład `User` i `Group`. Natomiast modele tworzone przez programistę muszą być rejestrowane ręcznie.

Kiedy rozpoczęliśmy pracę nad aplikacją `learning_logs`, framework Django utworzył plik o nazwie `admin.py` w tym samym katalogu, w którym znajduje się plik `models.py`.

*Plik admin.py:*

---

```
from django.contrib import admin

# Miejsce na rejestrację modeli.
```

---

Aby zarejestrować model `Topic` w witrynie administracyjnej, zmodyfikuj plik `admin.py` w poniższy sposób:

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

W powyższym fragmencie kodu mamy polecenie `from .models import Topic`, za pomocą którego importujemy model przeznaczony do zarejestrowania modelu `Topic`. Kropka przed słowem `models` nakazuje Django wyszukanie pliku o nazwie `models.py` znajdującego się w tym samym katalogu, w którym jest plik `admin.py`. Natomiast wywołanie `admin.site.register()` nakazuje framework'owi Django umożliwić zarządzanie tym modelem przez witrynę administracyjną.

Teraz konto superużytkownika wykorzystamy do uzyskania dostępu do witryny administracyjnej. W przeglądarce internetowej przejdź pod adres `http://localhost:8000/admin/`, podaj nazwę użytkownika i hasło utworzonego przed chwilą superużytkownika, a znajdziesz się na stronie podobnej do pokazanej na rysunku 18.2. Ta strona pozwala dodawać nowych użytkowników i grupy oraz wprowadzać zmiany w już istniejących listach użytkowników i grup. Zyskujemy także możliwość pracy z danymi powiązanymi ze zdefiniowanym wcześniej modelem `Topic`.



Rysunek 18.2. Witryna administracyjna wraz ze zdefiniowanym wcześniej modelem `Topic`

**UWAGA** Jeżeli w przeglądarce internetowej otrzymasz komunikat o niedostępności strony, upewnij się, że w powłoce nadal jest uruchomiony serwer Django. Jeżeli nie, aktywuj środowisko wirtualne i ponownie wydaj polecenie `python manage.py runserver`. W przypadku problemów z wyświetlением projektu na dowolnym etapie pracy zamknijcie okna powłoki oraz ponowne wydanie polecenia `runserver` jest dobrym pierwszym krokiem podczas rozwiązywania problemów.

## Dodanie tematu

Skoro model `Topic` został zarejestrowany w witrynie administracyjnej, możemy dodać pierwszy temat. Kliknij `Topics` i przejdź na stronę `Topics`, która w tym momencie jest pusta, ponieważ nie mamy jeszcze tematów, którymi moglibyśmy zarządzać. Kiedy klikniesz przycisk `Dodaj`, otrzymasz formularz pozwalający na dodanie nowego tematu. Wpisz **Szachy** i naciśnij przycisk `Zapisz`. Powrócisz na stronę `Topics`, na której tym razem będzie wyświetlony utworzony przed chwilą temat.

Utworzymy teraz drugi temat, aby mieć nieco więcej danych, z którymi będziemy pracować. Ponownie kliknij przycisk `Dodaj`, a następnie wpisz nazwę tematu, na przykład **Wspinaczka górska**. Kiedy klikniesz przycisk `Zapisz`, znowu znajdziesz się na stronie `Topics`, na której będą wymienione oba utworzone przed chwilą tematy.

## Zdefiniowanie modelu Entry

Aby użytkownik mógł zarejestrować postępy, jakie poczynił w nauce gry w szachy lub wspinaczce górskiej, konieczne jest zdefiniowanie modelu dla wpisów, za pomocą których użytkownik będzie dokumentował swoje postępy. Każdy wpis musi być powiązany z określonym tematem. Tego rodzaju powiązanie jest nazywane „związkiem typu wiele do jednego”, co oznacza, że z jednym tematem może być powiązanych wiele wpisów.

Poniżej przedstawiłem kod dla modelu `Entry`. Umieść go w pliku `models.py`.

*Plik models.py:*

---

```
from django.db import models

class Topic(models.Model):
    --cięcie--

class Entry(models.Model): ❶
    """Konkretnie informacje o postępie w nauce."""
    topic = models.ForeignKey(Topic, on_delete=models.CASCADE) ❷
    text = models.TextField() ❸
    date_added = models.DateTimeField(auto_now_add=True)

    class Meta: ❹
        verbose_name_plural = 'entries'

    def __str__(self):
        """Zwraca reprezentację modelu w postaci ciągu tekstowego."""
        return f'{self.text[:50]}...' ❺
```

---

Klasa `Entry` dziedziczy po klasie bazowej Django o nazwie `Model`, podobnie jak klasa `Topic` (patrz wiersz ❶). Pierwszy atrybut (`topic`) to egzemplarz `ForeignKey` (patrz wiersz ❷), czyli *klucz zewnętrzny*. Klucz zewnętrzny to pojęcie z terminologii stosowanej w bazach danych, które oznacza odwołanie do innego rekordu w bazie danych. W omawianym przykładzie jest to polecenie łączące każdy wpis z określonym tematem. Każdemu utworzonemu tematowi został przypisany *klucz*

(inaczej identyfikator). Kiedy Django musi utworzyć połączenie między dwoma fragmentami danych, wówczas wykorzystuje powiązane z nimi klucze. Tego rodzaju połączenia będziemy wkrótce stosować do pobierania wszystkich wpisów powiązanych z danym tematem. Argument `on_delete=models.CASCADE` wskazuje Django, że po usunięciu danego tematu wszystkie powiązane z nim wpisy również powinny zostać usunięte. Jest to określone mianem *usunięcia kaskadowego*.

Następny atrybut nosi nazwę `text` i jest egzemplarzem `TextField` (patrz wiersz ③). Ten rodzaj danych nie narzuca żadnych ograniczeń dotyczących wielkości, ponieważ nie chcemy ograniczać długości poszczególnych wpisów. Atrybut `date_added` pozwala wyświetlać wpisy w kolejności ich tworzenia oraz umieszczać znacznik czasu obok każdego wpisu.

W wierszu ④ mamy klasę `Meta` zagnieźdzoną wewnątrz klasy `Entry`. Klasa `Meta` przechowuje informacje dodatkowe przydatne podczas zarządzania modelem. W omawianym przykładzie zdefiniowaliśmy atrybut specjalny nakazujący Django użycie formy `entries` podczas odwoływania się do więcej niż tylko jednego wpisu. Jeżeli nie zdefiniowalibyśmy tego atrybutu, to w przypadku konieczności odwołania się do wielu wpisów framework Django używałby formy `Entries`.

Metoda `_str_()` wskazuje, które informacje mają zostać wyświetlone podczas odwoływania się do poszczególnych wpisów. Ponieważ cały wpis może mieć postać dość długiego tekstu, tutaj nakazujemy wyświetlenie jedynie pierwszych 50 znaków przechowywanych przez atrybut `text` (patrz wiersz ⑤). Ponadto dołączamy również wielokropki, aby tym samym wyraźnie wskazać, że nie został wyświetlony pełny wpis.

## Migracja modelu `Entry`

Ponieważ dodaliśmy nowy model, musimy ponownie przeprowadzić migrację bazy danych. Ten proces będzie przebiegał podobnie jak wcześniej: zmodyfikujemy plik `models.py`, wydamy polecenie `python manage.py makemigrations nazwa_aplikacji`, a następnie polecenie `python manage.py migrate`.

Przeprowadź migrację bazy danych i sprawdź wygenerowane dane wyjściowe:

```
(11_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  learning_logs/migrations/0002_entry.py: ①
    - Create model Entry
(11_env)learning_log$ python manage.py migrate
Operations to perform:
  --cięcie--
  Applying learning_logs.0002_entry... OK ②
```

Wygenerowana została nowa migracja o nazwie `0002_entry.py`, zawierająca informacje o modyfikacji bazy danych w taki sposób, aby mogła ona przechowywać dane powiązane z modelem `Entry` (patrz wiersz ①). Po wydaniu polecenia `migrate` widzimy, że framework Django przeprowadził migrację, która zakończyła się powodzeniem (patrz wiersz ②).

## Rejestracja modelu Entry w witrynie administracyjnej

Model Entry również trzeba zarejestrować. Poniżej przedstawiłem zawartość pliku *admin.py* po wprowadzonej zmianie.

Plik *admin.py*:

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Jeżeli ponownie przejdziesz pod adres *http://localhost/admin/*, powinieneś zobaczyć pozycję *Entries* wymienioną w *learning\_logs*. Kliknij łącze *Dodaj* wyświetlane obok *Entries* lub najpierw kliknij *Entries*, a dopiero później *Dodaj wpis*. Powinieneś zobaczyć rozwijaną listę tematów, dla których można utworzyć wpis, oraz pole tekstowe przeznaczone na treść wpisu. Z rozwijanej listy wybierz temat *Szachy* i dodaj wpis. Poniżej przedstawiłem pierwszy dodany przez mnie wpis:

Otwarcie to pierwsza część gry, mniej więcej pierwsze dziesięć ruchów. Podczas otwarcia dobrze jest wykonać trzy zadania: ruszyć się gońcami i skoczkami, spróbować przejąć kontrolę nad częścią środkową szachownicy oraz zabezpieczyć króla.

Oczywiście to są tylko wskazówki. Trzeba koniecznie nauczyć się oceniać, kiedy stosować się do wskazówek, a kiedy zupełnie je ignorować.

Kiedy klikniesz przycisk *Zapisz*, powrócisz na główną stronę administracyjną dla wpisów. Od razu przekonasz się o korzyściach płynących z wyświetlenia reprezentacji danego wpisu jako ciągu tekstowego zawierającego jedynie pierwszych 50 znaków (`text[:50]`). Praca z wieloma wpisami w interfejsie witryny administracyjnej będzie znacznie łatwiejsza, gdy będziesz widział tylko początek wpisu zamiast jego pełnej treści.

Utwórz drugi wpis dotyczący szachów oraz jeden dotyczący wspinaczki górskiej. W ten sposób będziemy mieli pewne dane początkowe. Poniżej przedstawiłem drugi wpis odnoszący się do gry w szachy:

W początkowej fazie rozgrywki bardzo ważne jest wykonanie ruchów gońcami i skoczkami. Te figury mają potężne możliwości i są na tyle zwrotne, że odgrywają istotne role w pierwszych ruchach w grze.

Natomiast poniżej znajduje się pierwszy wpis o wspinaczce górskiej:

Jedną z najważniejszych umiejętności potrzebnych we wspinaczce górskiej jest umiejętność przenoszenia ciężaru ciała na stopy. Można się spotkać z błędym przekonaniem, że alpinista może przez cały dzień wisieć na rękach.

W rzeczywistości dobrzy alpiniści opracowali konkretne sposoby, które pozwalają im jak najczęściej przenosić ciężar na stopy, gdy tylko istnieje taka możliwość.

Te trzy wpisy to nasze dane początkowe, które wykorzystamy podczas dalszej rozbudowy aplikacji *Learning Log*.

## Powłoka Django

Po wprowadzeniu pewnych danych możemy je przeanalizować w sposób programowy za pomocą interaktywnej sesji w powłoce. To środowisko interaktywne, nazywane **powłoką** Django, jest doskonałym miejscem do testowania projektu oraz rozwiązywania ewentualnych problemów. Poniżej przedstawiłem przykład interaktywnej sesji w powłoce:

---

```
(11_env)learning_log$ python manage.py shell
>>> from learning_logs.models import Topic ❶
>>> Topic.objects.all()
<QuerySet [, <Topic: Wspinaczka górska>]>
```

---

Polecenie `python manage.py shell` (wydaj je w aktywnym środowisku wirtualnym) powoduje uruchomienie interpretera Pythona, za pomocą którego można analizować i przeglądać dane przechowywane w bazie danych projektu. W powyższym fragmencie kodu zaimportowaliśmy model `Topic` z modułu `learning_logs.models` (patrz wiersz ❶). Następnie wykorzystaliśmy metodę `Topic.objects.all()` do pobrania wszystkich egzemplarzy modelu `Topic`. Lista zwrócona w wyniku tego wywołania nosi nazwę *queryset*.

Przez otrzymany zbiór wyników (wspomniany wcześniej *queryset*) możemy przeprowadzić iterację w dokładnie taki sam sposób jak przez listę. Poniższy fragment kodu wyświetla identyfikatory przypisane poszczególnym obiektom tematów:

---

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...     print(topic.id, topic)
...
1 Szachy
2 Wspinaczka górska
```

---

Zbiór wyników umieszczamy w zmiennej `topics`, a następnie wyświetlamy wartość atrybutu `id` każdego tematu oraz przedstawioną w postaci ciągu tekstu-wego reprezentację każdego tematu. Jak możesz zobaczyć w wygenerowanych danych wyjściowych, szachy mają identyfikator 1, natomiast wspinaczka górska identyfikator 2.

Jeżeli chcesz poznać identyfikator określonego obiektu, możesz go pobrać za pomocą metody `Topic.objects.get()`, a później przeanalizować wszystkie jego atrybuty. Spójrzmy na wartości atrybutów `text` i `date_added` dla obiektu przedstawiającego temat gry w szachy:

---

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Szachy'
>>> t.date_added
datetime.datetime(2022, 5, 20, 3, 33, 36, 928759,
tzinfo=datetime.timezone.utc)
```

---

Istnieje również możliwość wyszukania wpisów powiązanych z określonym tematem. Wcześniej zdefiniowaliśmy atrybut `topic` w modelu `Entry`. Wymieniony atrybut jest tak zwanym kluczem zewnętrznym (`ForeignKey`), czyli rodzajem połączenia między nowym wpisem i tematem. Django wykorzysta to połączenie do pobrania każdego wpisu, który jest powiązany ze wskazanym tematem, tak jak pokazałem poniżej:

---

```
>>> t.entry_set.all() ❶
<QuerySet [

---


```

Aby pobrać dane za pomocą klucza zewnętrznego, nazwę modelu należy podać zapisaną małymi literami, a następnie dodać znak podkreślenia i słowo `set` (patrz wiersz ❶). Na przykład przyjmujemy założenie, że istnieją modele `Pizza` i `Topping`, przy czym model `Topping` jest powiązany z `Pizza` za pomocą klucza zewnętrznego. Kiedy utworzony obiekt będzie nosił nazwę `my_pizza` i będzie reprezentował pojedynczą pizzę, wówczas wszystkie dodatki do pizzy będzie można pobrać za pomocą wywołania `my_pizza.topping_set.all()`.

Tego rodzaju składnię wykorzystamy, gdy zaczniemy tworzyć kod stron, które będą mogły być wyświetlane przez użytkowników. Powłoka jest bardzo użyteczna, kiedy chcemy sprawdzić, czy kod pobiera dane zgodnie z oczekiwaniami. Gdy kod działa w powłoce zgodnie z oczekiwaniami, wówczas będzie działał równie dobrze w plikach tworzonych w projekcie. Natomiast gdy sprawdzany kod generuje błędy lub nie pobiera żądanych danych, wówczas łatwiej będzie poprawić te błędy z poziomu prostego środowiska powłoki niż z poziomu plików generujących strony internetowe. Nie będziemy się zbyt często odwoływać do powłoki, ale powinieneś regularnie jej używać, aby ćwiczyć pracę ze składnią Django dotyczącą uzyskiwania dostępu do danych przechowywanych w projekcie.

Za każdym razem, gdy zmodyfikujesz modele, będziesz musiał ponownie uruchomić powłokę, aby zobaczyć efekty wprowadzonych zmian. By zakończyć sesję powłoki, naciśnij klawisze `Ctrl+D` (w systemie Windows naciśnij `Ctrl+Z`, a później `Enter`).

## ZRÓB TO SAM

**18.2. Krótkie wpisy.** Metoda `_str_()` w modelu `Entry` aktualnie powoduje dołączenie wielokropka do każdego egzemplarza klasy `Entry`, gdy Django wyświetla go w witrynie administracyjnej bądź w powłoce. Do wymienionej metody dodaj polecenie `if`, które spowoduje dodanie wielokropka tylko dla wpisów dłuższych niż 50 znaków. Za pomocą witryny administracyjnej dodaj wpis krótszy niż 50 znaków i sprawdź, czy w trakcie jego wyświetlania jest dołączany wielokropki.

**18.3. API Django.** Podczas tworzenia kodu pozwalającego uzyskać dostęp do danych w projekcie przygotowujesz tak zwane *zapytanie*. Przejrzyj dostępną na stronie <https://docs.djangoproject.com/en/4.1/topics/db/queries/> dokumentację dotyczącą tworzenia zapytań pobierających dane. Większość informacji na podanej stronie będzie dla Ciebie nowością, ale staną się użyteczne, gdy zaczniesz pracować nad własnymi projektami.

**18.4. Pizzeria.** Utwórz nowy projekt o nazwie `pizzeria` wraz z aplikacją o nazwie `pizzas`. Zdefiniuj model `Pizza` wraz z atrybutem `name`, który będzie przechowywał wartości takie jak hawajska lub mięsna. Zdefiniuj drugi model o nazwie `Topping` wraz z atrybutami `pizza` i `name`. Atrybut `pizza` powinien być kluczem zewnętrznym dla modelu `Pizza`, natomiast atrybut `name` powinien przechowywać wartości takie jak ananas, bekon i sos.

Zarejestruj oba modele w witrynie administracyjnej. W tej witrynie wprowadź pewne nazwy pizzy i dodatków. Za pomocą powłoki przejrzyj wprowadzone dane.

# Tworzenie stron internetowych — strona główna aplikacji

Przygotowanie strony internetowej w Django to proces, który najczęściej składa się z trzech etapów: zdefiniowania adresu URL, utworzenia widoku i zbudowania szablonu. Przede wszystkim konieczne jest zdefiniowanie wzorca dla adresu URL. Każdy *wzorzec URL* opisuje układ adresu URL i informuje Django, na co należy zwrócić uwagę podczas dopasowywania żądania przeglądarki internetowej do adresu URL witryny internetowej. W ten sposób framework będzie wiedział, która strona internetowa powinna zostać zwrócona.

Każdy adres URL jest mapowany na określony *widok*, a funkcja widoku pobiera i przetwarza dane niezbędne dla konkretnej strony. Funkcja widoku często wywołuje tak zwany *szablon* — na jego podstawie jest budowana strona, która następnie zostanie przekazana przeglądarce internetowej. Aby pokazać, jak to działa w praktyce, przygotujemy teraz stronę główną aplikacji *Learning Log*. Zdefiniujemy adres URL dla strony głównej, utworzymy jej funkcję widoku oraz prosty szablon.

Ponieważ naszym celem jest zapewnienie tego, aby aplikacja *Learning Log* działała zgodnie z oczekiwaniami, w tym momencie utworzymy jedynie bardzo prostą stronę. Nadawanie stylu ukończonej i poprawnie funkcjonującej aplikacji internetowej przynosi wiele radości, jednak tworzenie aplikacji, która będzie wyglądać dobrze, ale nie będzie działać zgodnie z oczekiwaniami, jest bezcelowe. Na obecnym etapie strona główna będzie wyświetlała jedynie tytuł i krótki opis.

## Mapowanie adresu URL

Użytkownik żąda wyświetlenia strony internetowej przez wpisanie jej adresu URL w przeglądarce internetowej oraz przez klikanie łączy. Konieczne jest więc ustalenie, jakie adresy URL są niezbędne w projekcie. Zaczynamy od adresu URL strony głównej — to będzie bazowy adres URL podawany przez użytkownika, aby uzyskać dostęp do projektu. Obecnie bazowy adres URL w postaci `http://localhost:8000/` prowadzi do domyślnej witryny Django informującej o prawidłowej konfiguracji projektu. Zmienimy to, mapując ten bazowy adres URL na stronę główną aplikacji *Learning Log*.

W katalogu głównym projektu (`ll_project`) otwórz plik `urls.py`. Zobaczysz poniższy fragment kodu.

*Plik ll\_project/urls.py:*

---

```
from django.contrib import admin ❶
from django.urls import path

urlpatterns = [ ❷
    path('admin/', admin.site.urls), ❸
]
```

---

Pierwsze dwa polecenia importują funkcje i moduły przeznaczone do zarządzania adresami URL w witrynie administracyjnej (patrz wiersz ❶). Następnie mamy zdefiniowaną zmienną o nazwie `urlpatterns` (patrz wiersz ❷). W pliku `urls.py` dla projektu jako całości zmienna `urlpatterns` zawiera zbiór adresów URL z aplikacji tworzących dany projekt. Polecenie w wierszu ❸ obejmuje moduł `admin.site.urls`, który definiuje wszystkie adresy URL, jakie mogą być żądane z witryny administracyjnej.

Musimy w tym miejscu dodać adres URL dla aplikacji `learning_logs`:

---

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

---

Zimportowaliśmy funkcję `include()` i dodaliśmy polecenie dołączające moduł `learning_logs.urls`.

Domyślny plik `urls.py` znajduje się w katalogu `ll_project`. Teraz musimy utworzyć drugi plik `urls.py`, tym razem w katalogu `learning_logs`. Utwórz więc nowy plik Pythona i zapisz go jako `urls.py` w katalogu `learning_logs`, a potem umieść w nim następujący kod.

#### Plik `learning_logs/urls.py`:

---

```
"""Definiuje wzorce adresów URL dla learning_logs.""" ❶
from django.urls import path ❷
from . import views ❸
app_name = 'learning_logs' ❹
urlpatterns = [ ❺
    # Strona główna.
    path('', views.index, name='index'), ❻
]
```

---

Aby nie było wątpliwości, z którym plikiem `urls.py` pracujemy, na początku pliku umieściliśmy odpowiedni komunikat typu *docstring* (patrz wiersz ❶). W wierszu ❷ importujemy funkcję `path()`, która będzie niezbędna do mapowania adresów URL na widoki. Importujemy także moduł `views` (patrz wiersz ❸); kropka w tym poleceniu nakazuje Pythonowi zimportowanie widoków z tego samego katalogu, w którym znajduje się bieżący moduł `urls.py`. Zmienna `app_name` pomaga Django w odróżnieniu tego pliku `urls.py` od pliku o takiej samej nazwie znajdującego się w innych aplikacjach projektu (patrz wiersz ❹). Zmienna `urlpatterns` w tym module to lista poszczególnych stron, które mogą być żądane od aplikacji `learning_logs` (patrz wiersz ❺).

Rzeczywistym wzorcem URL jest wywołanie funkcji `path()`, które pobiera trzy argumenty (patrz wiersz ❻). Pierwszy to ciąg tekstowy pomagający Django we właściwym przekierowaniu żądania. Django otrzymuje żądany adres URL i próbuje przekazać to żądanie do widoku. Odbywa się to przez sprawdzenie wszystkich zdefiniowanych wzorców URL w celu znalezienia tego, który można dopasować do bieżącego żądania. Django ignoruje bazowy adres URL dla projektu (`http://localhost:8000/`), więc pusty ciąg tekstowy ('') jest dopasowywany do bazowego adresu URL. Żaden inny adres URL nie zostanie dopasowany do tego wzorca, a Django zwróci stronę błędu, jeśli żądany adres URL nie został dopasowany do istniejących wzorców URL.

Drugi argument w wywołaniu funkcji `path()` w wierszu ❻ wskazuje funkcję widoku w pliku `views.py` przeznaczoną do użycia. Kiedy żądany adres URL zostanie dopasowany do zdefiniowanego wzorca, Django wywoła `index()` z pliku `views.py` (tę funkcję widoku utworzymy w następnej sekcji). Trzeci argument wskazuje nazwę `index` dla danego wzorca adresu URL — za pomocą tej nazwy

możemy odwoływać się do zdefiniowanego wzorca z poziomu innych komponentów aplikacji. Jeśli kiedykolwiek będziesz chciał dostarczyć łącze do strony głównej, będziesz używać zdefiniowanej nazwy, zamiast podawać adres URL.

## Utworzenie widoku

Funkcja widoku pobiera informacje z żądania, przygotowuje dane niezbędne do wygenerowania strony, a następnie przekazuje je do przeglądarki internetowej, często używając przy tym szablonu, który definiuje wygląd strony.

Plik `views.py` w katalogu `learning_logs` został wygenerowany automatycznie po wydaniu polecenia `python manage.py startapp`. Poniżej przedstawiłem obecną zawartość pliku `views.py`.

*Plik views.py:*

---

```
from django.shortcuts import render

# Miejsce na utworzenie widoków.
```

---

Aktualnie ten plik jedynie importuje funkcję `render()`, która jest odpowiedzialna za wygenerowanie odpowiedzi na podstawie danych dostarczonych przez widoki. Poniższy fragment kodu pokazuje, jak należy utworzyć widok dla strony głównej budowanej aplikacji:

---

```
from django.shortcuts import render

def index(request):
    """Strona główna dla aplikacji Learning Log."""
    return render(request, 'learning_logs/index.html')
```

---

Kiedy adres URL żądania zostanie dopasowany do zdefiniowanego wzorca, Django będzie szukać w pliku `views.py` funkcji o nazwie `index()`. Funkcji tej zostanie przekazany obiekt `request`. W omawianym przypadku nie ma potrzeby przetwarzania jakichkolwiek danych dla strony, więc jedyny kod w tej funkcji to wywołanie `render()`. Użyta tutaj funkcja `render()` ma dwa argumenty: pierwotny obiekt `request` i szablon, który zostanie wykorzystany do przygotowania strony. Przystępujemy teraz do utworzenia szablonu.

## Utworzenie szablonu

Szablon definiuje wygląd strony, a Django umieszcza na niej odpowiednie dane za każdym razem po otrzymaniu żądania do strony. Dzięki szablonowi można uzyskać dostęp do wszelkich danych dostarczanych przez widok. Ponieważ przygotowany wcześniej widok dla strony głównej nie dostarcza żadnych danych, szablon ten jest całkiem prosty.

W katalogu `learning_logs` utwórz nowy podkatalog o nazwie `templates`. Następnie w katalogu `templates` utwórz podkatalog o nazwie `learning_logs`. Taka struktura katalogów może wydawać się niepotrzebna (katalog `learning_logs` zawiera podkatalog `templates`, który z kolei zawiera kolejny katalog o nazwie `learning_logs`), ale będzie przez Django interpretowana bez problemów, nawet w przypadku ogromnego projektu zawierającego wiele pojedynczych aplikacji. W drugim podkatalogu `learning_logs` utwórz plik o nazwie `index.html`. Ścieżka dostępu do tego pliku będzie miała postać `ll_project/learning_logs/templates/learning_logs/index.html`. Następnie w podanym pliku umieść poniższy fragment kodu.

**Plik index.html:**

---

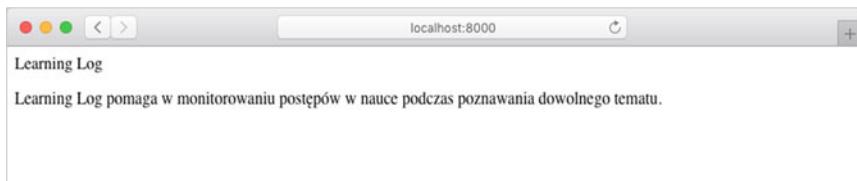
```
<p>Learning Log</p>

<p>Learning Log pomaga w monitorowaniu postępów w nauce podczas poznawania dowolnego tematu.</p>
```

---

To jest bardzo prosty plik. Jeżeli nie masz zbyt dużego doświadczenia w tworzeniu kodu HTML, to wyjaśniam, że znaczniki `<p></p>` definiują akapit. Znacznik `<p>` otwiera akapit, natomiast `</p>` go zamyka. W powyższym fragmencie kodu mamy dwa akapity: pierwszy jest rodzajem tytułu, natomiast drugi wyjaśnia użytkownikowi przeznaczenie aplikacji *Learning Log*.

Teraz po wykonaniu żądania do bazowego adresu URL projektu, czyli `http://localhost:8000/`, zamiast strony domyślnie generowanej przez Django zobaczysz wyświetlzoną stronę zbudowaną powyżej. Framework pobiera żądany adres URL i po dopasowaniu go do wzorca '' wywołuje funkcję `views.index()`, która z kolei generuje stronę za pomocą szablonu zdefiniowanego w pliku `index.html`. Otrzymaną stronę pokazałem na rysunku 18.3.



Rysunek 18.3. Strona główna aplikacji Learning Log

Wprawdzie proces tworzenia jednej strony może wydawać się skomplikowany, ale w rzeczywistości rozdzielenie tych trzech etapów — zdefiniowania adresów URL, utworzenia widoków i zbudowania szablonów — sprawdza się doskonale. Dzięki temu możemy skupić się na poszczególnych aspektach projektu. Natomiast w dużych projektach ułatwia to programistom skoncentrowanie się na zadaniach, w których są najlepsi. Na przykład specjalista od baz danych może skoncentrować się na modelach, programista na kodzie widoku, natomiast projektant WWW na szablonach.

**UWAGA** *Możesz otrzymać następujący komunikat błędu:*

---

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

---

*W takim przypadku zatrzymaj serwer programistyczny przez naciśnięcie klawiszy Ctrl+C w powłoce, w której zostało wydane polecenie runserver. Potem ponownie wydaj polecenie python manage.py runserver. Powinieneś zobaczyć stronę główną. Jeżeli kiedykolwiek napotkasz ten błąd, spróbuj ponownie uruchomić serwer programistyczny.*

### ZRÓB TO SAM

**18.5. Jadłospis.** Rozważ aplikację ułatwiającą użytkownikom przygotowanie jadłospisu na cały tydzień. Utwórz nowy katalog o nazwie *meal\_planner* i rozpoczęj w nim pracę nad nowym projektem Django. Następnie utwórz nową aplikację o nazwie *meal\_plans*. Przygotuj prostą stronę główną dla tego projektu.

**18.6. Strona główna pizzerii.** Dodaj stronę główną do projektu pizzerii, nad którym pracę rozpoczęłeś w ćwiczeniu 18.4.

## Utworzenie dodatkowych stron

Skoro opracowaliśmy procedurę tworzenia strony, możemy przystąpić do budowania projektu *Learning Log*. Potrzebujemy dwóch kolejnych stron do wyświetlania danych. Pierwsza z nich będzie zawierała listę wszystkich tematów, natomiast druga będzie wyświetlała listę wszystkich wpisów dla poszczególnych tematów. Dla każdej z tych stron zdefiniujemy wzorzec adresów URL, przygotujemy funkcję widoku oraz szablon. Jednak zanim do tego przejdziemy, zajmiemy się utworzeniem szablonu bazowego, po którym będą dziedziczyć wszystkie szablony projektu.

### Dziedziczenie szablonu

Kiedy budujemy witryny internetowe, niemal zawsze się zdarza, że pewne elementy powtarzają się na wszystkich stronach. Zamiast umieszczać je bezpośrednio na każdej stronie, lepiej zrobimy, jeśli przygotujemy szablon bazowy zawierający powtarzające się elementy, a następnie zastosujemy na nich dziedziczenie po tym szablonie. Takie podejście pozwala skoncentrować się na unikatowych aspektach poszczególnych stron, a ponadto bardzo ułatwia zmianę ogólnego widoku i sposobu działania projektu.

### Szablon nadzędny

Rozpoczynamy od utworzenia szablonu o nazwie *base.html* i umieszczamy go w tym samym katalogu, w którym znajduje się już plik *index.html*. Ten szablon bazowy będzie zawierał elementy stosowane na wszystkich stronach, a pozostałe

szablony będą dziedziczyły po *base.html*. Obecnie jedynym elementem, który chcemy umieścić na każdej stronie, jej tytuł wyświetlny na jej początku. Ponieważ omawiany szablon zostanie dołączony do każdej strony, umieszczać w nim tytuł i łącze prowadzące do strony głównej.

**Plik *base.html*:**

---

```
<p>
    <a href="{% url 'learning_logs:index' %}">Learning Log</a> ①
</p>

{% block content %}{% endblock content %} ②
```

---

W pierwszej części tego pliku tworzymy akapit zawierający nazwę projektu, która jednocześnie jest łączem do strony głównej. Aby wygenerować łącze, używamy tak zwanego *szablonu znacznika*, na co wskazują nawiasy klamrowe i znaki procentu — { % }. Szablon znacznika to fragment kodu generujący informacje przeznaczone do wyświetlenia na stronie. W omawianym przykładzie szablon znacznika { % url 'learning\_logs:index' %} powoduje wygenerowanie adresu URL dopasowującego wzorzec URL o nazwie 'index' zdefiniowany w pliku *learning\_logs/urls.py* (patrz wiersz ①). Tutaj *learning\_logs* to *przestrzeń nazw*, natomiast *index* to unikatowa nazwa wzorca URL w tej przestrzeni nazw. Przestrzeń nazw pochodzi z wartości, która została przypisana zmiennej *app\_name* w pliku *learning\_logs/urls.py*.

W budowanej tutaj prostej stronie HTML łącze znajduje się w znaczniku *<a>*:

---

```
<a href="link_url">tekst łącza</a>
```

---

Posiadanie szablonu znacznika generującego adres URL bardzo ułatwia zachowanie aktualności łączy. W celu zmiany adresu URL w projekcie konieczne jest wprowadzenie modyfikacji jedynie we wzorcu URL zdefiniowanym w pliku *urls.py*, a Django automatycznie wstawi aktualny adres URL w trakcie kolejnego żądania strony. Ponieważ każda strona w budowanym projekcie będzie dziedziczyć po *base.html*, od teraz wszystkie strony będą miały łącze prowadzące na stronę główną.

W wierszu ② mamy parę znaczników *block*. Blok o nazwie *content* jest miejscem zarezerwowanym. Szablon potomny będzie definiował, jakiego rodzaju informacje zostaną umieszczone w bloku *content*.

Szablon potomny nie musi definiować każdego bloku wymienionego w szablonie nadzędnym. Dlatego też w szablonach nadzędnych możesz przygotować dowolną liczbę bloków, natomiast w szablonach potomnych zostaną użyte jedynie te wymagane na danej stronie.

**UWAGA** W kodzie Pythona niemal zawsze stosujemy wcięcia o wielkości czterech spacji. Ponieważ pliki szablonów zwykle mają więcej poziomów wcięć niż plik kodu źródłowego Pythona, w szablonach przyjęto się stosowanie wcięć o wielkości tylko dwóch spacji.

## Szablon potomny

W kolejnym kroku zmodyfikujemy plik szablonu *index.html* w taki sposób, aby dziedziczył po *base.html*. Poniżej przedstawiłem aktualną zawartość pliku *index.html*.

Plik *index.html*:

---

```
{% extends "learning_logs/base.html" %} ❶

{% block content %} ❷
    <p>Learning Log pomaga w monitorowaniu postępów w nauce.</p>
{% endblock content %} ❸
```

---

Jeżeli porównasz powyższy kod z pierwotną zawartością pliku *index.html*,auważysz zastąpienie tytułu „Learning Log” kodem wskazującym na dziedziczenie po szablonie nadzędnym (patrz wiersz ❶). Szablon potomny musi zawierać znacznik `{% extends %}` w pierwszym wierszu, aby tym samym wskazać frameworkowi Django, po którym szablonie nadzędnym dziedziczy. Plik *base.html* jest częścią aplikacji *learning\_log* i dlatego podaliśmy nazwę tej aplikacji w ścieżce dostępu wskazującej szablon. Ten wiersz pobiera cały kod z szablonu *base.html* i pozwala szablonowi *index.html* na zdefiniowanie zawartości umieszczonej w miejscu zarezerwowanym przez blok *content*.

Blok treści został zdefiniowany w wierszu ❷ przez wstawienie znacznika `{% block %}` wraz z nazwą *content*. Wszystko to, co nie jest dziedziczone po szablonie nadzędnym, zostaje umieszczone wewnątrz bloku *content*. W omawianym przykładzie mamy akapit opisujący projekt *Learning Log*. W wierszu ❸ za pomocą znacznika `{% endblock content %}` wskazujemy na zakończenie definiowania bloku treści. Wprawdzie znacznik `{% endblock %}` nie wymaga nazwy, ale jeśli szablon stanie się znacznie większy i będzie zawierał wiele bloków, wówczas dobrze jest wiedzieć, który dokładnie blok został zakończony przez dany znacznik.

Możesz zacząć dostrzegać korzyści płynące z dziedziczenia szablonów — w szablonie potomnym trzeba umieścić jedynie tę treść, która jest unikatowa dla danej strony. W ten sposób nie tylko upraszczamy wszystkie szablony, ale jednocześnie znacznie ułatwiamy sobie przeprowadzanie ewentualnych zmian w witrynie. Aby zmodyfikować element wyświetlany na wielu stronach, zmianę trzeba wprowadzić tylko jednokrotnie — w elemencie znajdującym się w szablonie nadzędnym. Taka zmiana będzie później odzwierciedlona na każdej stronie dziedziczącej po zmodyfikowanym szablonie. W projekcie składającym się z dziesiątek lub setek stron, tego rodzaju struktura znacznie ułatwia usprawnienie witryny i jednocześnie skraca czas potrzebny na wprowadzenie zmian.

W dużych projektach dość często można się spotkać z istnieniem jednego szablonu nadzędneg o nazwie *base.html*, który jest przeznaczony dla całej witryny internetowej, oraz szablonów nadzędnych, które są przeznaczone dla najważniejszych sekcji witryny. Wszystkie szablony sekcji dziedziczą po *base.html*, a każda strona w witrynie dziedziczy po szablonie sekcji. W ten sposób można

dość łatwo modyfikować wygląd i sposób działania witryny internetowej jako całości, jej poszczególnych sekcji lub stron. Tego rodzaju konfiguracja zapewnia bardzo efektywny sposób pracy i zachęca do uaktualniania witryny w późniejszym czasie.

## Strona tematów

Skoro opracowaliśmy mechanizm przeznaczony do tworzenia stron, możemy teraz przystąpić do przygotowania dwóch kolejnych stron. Pierwsza z nich to ogólna strona tematów, natomiast druga służy do wyświetlenia wpisów dla pojedynczego tematu. Strona tematów, na której będą wyświetlane wszystkie tematy utworzone przez użytkownika, jednocześnie będzie pierwszą stroną wykorzystującą pracę z danymi.

### Wzorzec URL dla strony tematów

Zaczynamy od zdefiniowania adresu URL dla strony tematów. Powszechnie jest wybieranie takiego adresu URL, aby odzwierciedlał rodzaj informacji przedstawiających na danej stronie. W omawianym przykładzie wykorzystamy słowo *topics* — wpisanie w przeglądarce adresu URL w postaci `http://localhost:8000/topics/` spowoduje przejście na stronę tematów. Poniżej przedstawiłem zmodyfikowaną wersję pliku *learning\_logs/urls.py*.

Plik *learning\_logs/urls.py*:

---

```
"""Definiuje wzorce adresów URL dla learning_logs."""
--cięcie--
urlpatterns = [
    # Strona główna.
    path('', views.index, name='index'),
    # Wyświetlenie wszystkich tematów.
    path('topics/', views.topics, name='topics'),
]
```

---

Po prostu wstawiliśmy `topics/` do argumentu ciągu tekstuowego, który został użyty w celu dopasowania adresu URL strony głównej. Kiedy Django analizuje żądany adres URL, ten wzorzec powoduje dopasowanie każdego adresu URL, który składa się z bazowego adresu URL i znajdującego się po nim słowa *topics*. Na końcu możesz podać lub pominąć ukośnik. Jednak po słownie *topics* nie powinno się nic znajdować, ponieważ w przeciwnym razie wzorzec nie zostanie dopasowany. Każde żądanie zawierające adres URL dopasowany do wzorca zostanie przekazane funkcji `topics()` zdefiniowanej w pliku *views.py*.

## Widok tematów

Funkcja `topics()` musi pobrać pewne informacje z bazy danych, a następnie przekazać je do szablonu. Poniżej przedstawiłem kod, który trzeba dodać do pliku *views.py*.

### Plik views.py:

---

```
from django.shortcuts import render

from .models import Topic ①

def index(request):
    --cięcie--

def topics(request): ②
    """Wyświetlenie wszystkich tematów."""
    topics = Topic.objects.order_by('date_added') ③
    context = {'topics': topics} ④
    return render(request, 'learning_logs/topics.html', context) ⑤
```

---

Zaczynamy od zainportowania modelu powiązanego z potrzebnymi nam danymi (patrz wiersz ①). Funkcja `topics()` wymaga tylko jednego parametru — obiektu `request` otrzymanego przez Django z serwera WWW (patrz wiersz ②). W wierszu ③ przygotowujemy zapytanie do bazy danych: interesują nas obiekty `Topic` posortowane według wartości atrybutu `date_added`. Otrzymany zbiór wynikowy przechowujemy w zmiennej o nazwie `topics`.

W wierszu ④ definiujemy kontekst przekazywany później do szablonu. Ten *kontekst* to po prostu słownik, w którym klucze są nazwami używanymi w szablonie w celu uzyskania dostępu do danych, natomiast wartości to dane przekazywane do szablonu. W omawianym przykładzie mamy tylko jedną parę klucz-wartość zawierającą zbiór tematów przeznaczonych do wyświetlenia na stronie. Kiedy budujemy stronę wykorzystującą dane, zmienną `context` przekazujemy do metody `render()` razem z obiektem `request` i ścieżką dostępu do szablonu (patrz wiersz ⑤).

## Szablon tematów

Szablon dla strony tematów otrzymuje słownik `context`, co pozwala szablonowi na użycie danych dostarczonych przez funkcję `topics()`. W katalogu zawierającym już plik `index.html` utwórz nowy plik o nazwie `topics.html`. Poniżej przedstawiam kod wyświetlający tematy w szablonie.

### Plik topics.html:

---

```
{% extends "learning_logs/base.html" %}

{% block content %}

<p>Tematy</p>

<ul> ①
    {% for topic in topics %} ②
        <li>{{ topic.text }}</li> ③
    {% endfor %}
</ul>
```

```
{% empty %} ④  
<li>Nie został jeszcze dodany żaden temat.</li>  
{% endfor %} ⑤  
</ul> ⑥  
  
{% endblock content %}
```

---

Zaczynamy od użycia znacznika `{% extends %}` w celu dziedziczenia szablonu po `base.html`, podobnie jak to było w przypadku poprzedniego szablonu. Następnie rozpoczynamy blok `content`. Na treść tej strony będzie składała się wypunktowana lista wszystkich tematów. W standardowym kodzie HTML taka wypunktowana lista jest nazywana *listą nieuporządkowaną* — na jej użycie wskazują znaczniki `<ul></ul>`. Początek listy tematów jest generowany przez wiersz ①.

W wierszu ② mamy kolejny szablon znacznika, odpowiednik pętli `for` — dzięki temu będziemy mogli przeprowadzić iterację przez listę `topics` otrzymaną ze słownika `context`. Kod użyty w szablonie pod wieloma względami różni się od kodu stosowanego w Pythonie. Python wykorzystuje wcięcia w celu wskazania wierszy zaliczających się do bloku pętli `for`. Natomiast w szablonie każda pętla `for` wymaga znacznika `{% endfor %}` wskazującego miejsce zakończenia pętli. Dlatego też w szablonie struktura pętli przedstawia się następująco:

---

```
{% for item in list %}  
    wykonanie dowolnego zadania na każdym elemencie  
{% endfor %}
```

---

Wewnątrz bloku pętli każdy temat zamieniamy na element listy. W celu wyświetlenia zmiennej w szablonie, konieczne jest ujęcie nazwy zmiennej w podwójny nawias klamrowy. Te nawiasy klamrowe nie zostaną wyświetcone na stronie, jedynie wskazują Django na użycie zmiennej szablonu. Dlatego widoczne w wierszu ③ polecenie `{{ topic.text }}` zostanie zastąpione przez wartość atrybutu `text` bieżącego tematu w trakcie każdej iteracji pętli. Znacznik HTML `<li></li>` oznacza *element listy*. Wszystko to, co zostanie umieszczone między tymi znacznikami, wewnątrz pary znaczników `<ul></ul>`, będzie wyświetlane na stronie jako element wypunktowanej listy.

W wierszu ④ używamy szablonu znacznika `{% empty %}`, który wskazuje frameworkowi Django, co ma zrobić, jeśli lista nie będzie zawierała żadnych elementów. W omawianym przykładzie zostanie wyświetlony komunikat o tym, że użytkownik jeszcze nie dodał żadnego tematu. Ostatnie dwa wiersze kodu to zamknięcie pętli `for` (patrz wiersz ⑤) i wypunktowanej listy (patrz wiersz ⑥).

Teraz możemy zmodyfikować szablon bazowy w taki sposób, aby zawierał łącze do strony wyświetlającej tematy. Do pliku `base.html` należy dodać następujący kod.

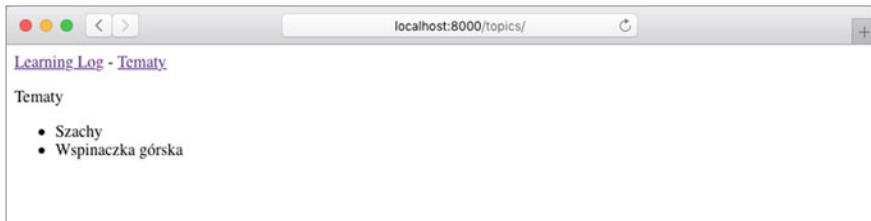
```
<p>
    <a href="{% url 'learning_logs:index' %}">Learning Log</a> - ❶
    <a href="{% url 'learning_logs:topics' %}">Tematy</a> ❷
</p>

{% block content %}{% endblock content %}
```

---

Po łączu prowadzącym do strony głównej umieściliśmy myślnik (patrz wiersz ❶), a następnie za pomocą znacznika szablonu `{% url %}` dodaliśmy łącze do strony wyświetlającej tematy (patrz wiersz ❷). Ten wiersz kodu nakazuje Django wygenerować łącze dopasowane do wzorca adresu URL o nazwie 'topic', zdefiniowanego w pliku `learning_logs/urls.py`.

Kiedy odświeżysz stronę główną w przeglądarce internetowej, zobaczysz łącze *Tematy*. Gdy jej klikniesz, zostanie wyświetlona strona podobna do pokazanej na rysunku 18.4.



Rysunek 18.4. Strona wyświetlająca tematy

## Strony poszczególnych tematów

Następnym krokiem jest utworzenie strony pozwalającej skoncentrować się na pojedynczym temacie. Taka strona ma wyświetlić nazwę tematu oraz wszystkie powiązane z nim wpisy. Ponownie zdefiniujemy nowy wzorzec adresu URL, utworzymy widok oraz szablon. Ponadto konieczne będzie zmodyfikowanie strony tematów, aby każdy element listy był łączem prowadzącym na odpowiednią stronę tematu.

### Wzorzec adresu URL dla tematu

Wzorzec adresu URL dla strony tematu jest nieco inny niż stosowane dotąd wzorce, ponieważ wykorzystuje atrybut `id` w celu wskazania żądanego tematu. Przykładowo jeśli użytkownik będzie chciał wyświetlić stronę zawierającą szczegóły tematu gry w szachy (wartość jego atrybutu `id` wynosi 1), wówczas adres URL będzie miał postać `http://localhost:8000/topics/1/`. Poniżej przedstawiłem wzorzec dopasowujący ten adres URL — należy go umieścić w pliku `learning_logs/urls.py`.

## Plik learning\_logs/urls.py:

---

```
--cięcie--  
urlpatterns = [  
    --cięcie--  
    # Strona szczegółowa dotycząca pojedynczego tematu.  
    path('topics/(<int:topic_id>)/', views.topic, name='topic'),  
]
```

---

Przeanalizujmy teraz ciąg tekstowy 'topics/(<int:topic\_id>)/' użyty w tym wzorcu URL. Część pierwsza tego ciągu tekstu nakazuje Django dopasowanie każdego adresu URL, który składa się z bazowego adresu URL i znajdującego się po nim słowa *topics*. Natomiast część druga tego ciągu tekstu, /<int:topic\_id>/, powoduje dopasowanie liczby całkowitej umieszczonej między dwoma ukośnikami. Ta wartość liczbową jest przechowywana w argumencie o nazwie *topic\_id*.

Kiedy Django znajdzie adres URL dopasowany do tego wzorca, wywołuje funkcję widoku o nazwie *topic()* wraz z wartością *topic\_id* jako argumentem. Tę wartość *topic\_id* wykorzystamy w celu pobrania w funkcji odpowiedniego tematu.

## Widok tematu

Funkcja *topic()* musi pobrać z bazy danych temat oraz wszystkie powiązane z nim wpisy, jak pokazałem w poniższym fragmencie kodu.

## Plik views.py:

---

```
--cięcie--  
def topic(request, topic_id): ❶  
    """Wyświetla pojedynczy temat i wszystkie powiązane z nim wpisy."""  
    topic = Topic.objects.get(id=topic_id) ❷  
    entries = topic.entry_set.order_by('-date_added') ❸  
    context = {'topic': topic, 'entries': entries} ❹  
    return render(request, 'learning_logs/topic.html', context) ❺
```

---

Jest to pierwsza funkcja widoku wymagająca podania parametru innego niż obiekt *request*. Ta funkcja akceptuje wartość przechwyconą przez wyrażenie /<int:topic\_id>/ i przechowuje ją w *topic\_id* (patrz wiersz ❶). W wierszu ❷ używamy funkcję *get()* w celu pobrania tematu, podobnie jak to wcześniej zrobiliśmy w trakcie sesji Django w powłoce. W wierszu ❸ pobieramy wpisy powiązane z danym tematem i układamy je w kolejności według wartości atrybutu *date\_added*. Znak minus na początku wymienionego atrybutu powoduje sortowanie w kolejności odwrotnej, więc ostatnio dodane wpisy będą wyświetlane jako pierwsze. Temat i powiązane z nim wpisy przechowujemy w słowniku *context* (patrz wiersz ❹) i wywołujemy metodę *render()* razem z obiektem *request*, szablonem *topic.html* i słownikiem *context* (patrz wiersz ❺).

**UWAGA** Fazy kodu w wierszach ❷ i ❸ są nazywane zapytaniami, ponieważ wykonujemy zapytanie do bazy danych o konkretne informacje. Kiedy we własnych projektach tworzysz tego rodzaju zapytania, najlepszym rozwiążaniem będzie ich wypróbowanie najpierw w Django. Pracując w powłoce, otrzymasz znacznie szybszą odpowiedź niż w przypadku tworzenia widoku i szablonu, a następnie sprawdzania w przeglądarce internetowej otrzymanych wyników.

## Szablon tematu

Szablon tematu powinien wyświetlać nazwę tematu oraz powiązane z nim wpisy. Ponadto jeśli do danego tematu nie został jeszcze dodany żaden wpis, użytkownik powinien zostać o tym poinformowany.

Plik topic.html:

---

```
{% extends 'learning_logs/base.html' %}

{% block content %}

<p>Temat: {{ topic.text }}</p> ❶

<p>Wpisy:</p>
<ul> ❷
    {% for entry in entries %} ❸
        <li>
            <p>{{ entry.date_added|date:'d M Y H:i' }}</p> ❹
            <p>{{ entry.text|linebreaks }}</p> ❺
        </li>
    {% empty %} ❻
        <li>Nie ma jeszcze żadnego wpisu.</li>
    {% endfor %}
</ul>

{% endblock content %}
```

---

Rozszerzamy szablon bazowy *base.html*, podobnie jak w przypadku wszystkich stron projektu. Następnym krokiem jest wyświetlenie aktualnego tematu (patrz wiersz ❶), którego nazwa jest przechowywana w atrybucie *text* żadanego szablonu. Zmienna *topic* jest dostępna, ponieważ została umieszczona w słowniku *context*. Następnie rozpoczynamy tworzenie wypunktowanej listy w celu wyświetlenia wszystkich wpisów (patrz wiersz ❷) oraz przeprowadzamy iterację przez te wpisy, podobnie jak to zrobiliśmy wcześniej w przypadku tematów (patrz wiersz ❸).

Każda wypunktowana lista zawiera dwa elementy informacji. Pierwszy to znacznik czasu, natomiast drugi to pełny tekst danego wpisu. W przypadku znacznika czasu (patrz wiersz ❹) wyświetlamy wartość atrybutu *date\_added*. W szablonach Django pionowa linia (|) oznacza *filtr* szablonu, czyli funkcję modyfikującą wartość w zmiennej szablonu. Filtr *date: 'd M Y H:i'* wyświetla znacznik

czasu w formacie *1 sty 2022 23:00*. W kolejnym wierszu zostaje wyświetlona pełna wartość `text`. Filtr `linebreaks` (patrz wiersz ⑤) gwarantuje, że długie wpisy nie zostaną wyświetlone w postaci ciągłego strumienia tekstu, tylko będą zawierały znaki nowego wiersza w formacie zrozumiałym dla przeglądarek internetowych. W wierszu ⑥ używamy znacznika szablonu `{% empty %}` do wyświetlenia użytkownikowi komunikatu o tym, że nie zostały dodane żadne wpisy.

## Łącza ze strony tematów

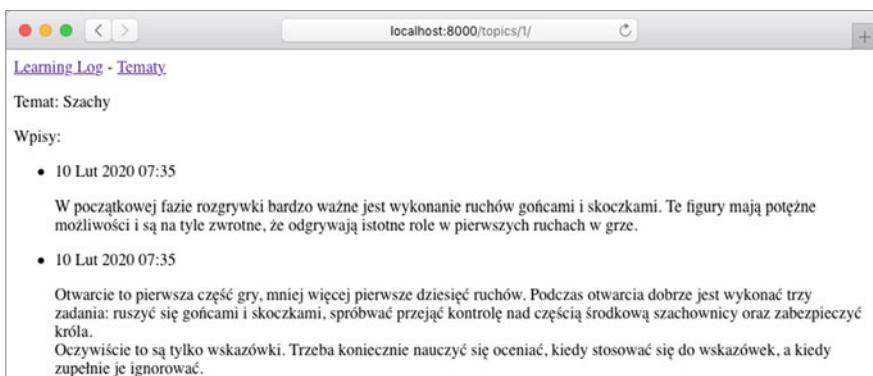
Zanim będzie można wyświetlić stronę danego tematu w przeglądarce internetowej, konieczne jest zmodyfikowanie szablonu tematów tak, aby każde łącze tematu prowadziło na odpowiednią stronę. Poniżej przedstawiłem zmiany, jakie należy wprowadzić w pliku `topics.html`.

Plik `topics.html`:

```
--cięcie--  
    {% for topic in topics %}  
        <li>  
            <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic.text }}</a>  
        </li>  
    {% empty %}  
--cięcie--
```

Szablon znacznika URL wykorzystujemy do wygenerowania odpowiedniego łącza na podstawie wzorca adresu URL w `learning_logs` o nazwie '`topic`'. Ten wzorzec adresu URL wymaga argumentu `topic_id`, więc dodajemy atrybut `topic.id` do szablonu znacznika URL. Teraz każdy temat na liście jest łączem, którego kliknięcie powoduje przejście na odpowiednią stronę, na przykład `http://localhost:8000/topics/1/`.

Jeżeli w przeglądarce internetowej odświeżysz stronę tematów i klikniesz wybrany temat, zobaczysz stronę podobną do tej pokazanej na rysunku 18.5.



Rysunek 18.5. Szczegółowa strona dla pojedynczego tematu — wyświetla wszystkie powiązane z nim wpisy

**UWAGA** Istnieje subtelna, choć jednocześnie ważna różnica między `topic.id` i `topic_id`. Wyrażenie `topic.id` analizuje temat i pobiera wartość odpowiadającego mu identyfikatora. Natomiast zmienna `topic_id` to odwołanie do tego identyfikatora w kodzie. Jeżeli podczas pracy z identyfikatorami wystąpią błędy, upewnij się, że wyrażenia tego używasz w odpowiedni sposób.

## ZRÓB TO SAM

**18.7. Dokumentacja szablonów.** Przejrzyj dokumentację szablonów Django, którą znajdziesz na stronie <https://docs.djangoproject.com/en/4.1/ref/templates/>. Zawsze możesz do niej powrócić, gdy będziesz pracować nad własnymi projektami.

**18.8. Strony pizzerii.** Do projektu pizzerii utworzonego w ćwiczeniu 18.6 dodaj stronę wyświetlającą nazwy dostępnych rodzajów pizzy. Następnie połącz nazwę każdej pizzy ze stroną wyświetlającą dostępne dodatki do pizzy. Upewnij się, że użyłeś dziedziczenia szablonów, aby efektywnie tworzyć strony.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak zacząć budować aplikacje internetowe za pomocą framework'a Django. Przygotowaliśmy krótką specyfikację projektu, zainstalowaliśmy framework'a Django w środowisku wirtualnym, skonfigurowaliśmy projekt oraz upewniliśmy się, że został przygotowany prawidłowo. Zobaczyłeś, jak skonfigurować aplikację i zdefiniować modele przedstawiające dane używane przez tę aplikację. Dowiedziałeś się nieco o bazach danych i zobaczyłeś, jak Django pomaga w migracji bazy danych po wprowadzeniu zmian w modelach. Dowiedziałeś się, jak utworzyć superużytkownika dla witryny internetowej oraz jak wykorzystać tę witrynę administracyjną do wprowadzenia pewnych danych początkowych.

Poznałeś również powłokę Django pozwalającą pracować z danymi projektu za pomocą sesji Django w terminalu. Zobaczyłeś, jak można zdefiniować adresy URL, utworzyć funkcje widoków oraz szablony, na podstawie których powstaną strony witryny internetowej. Na końcu nauczyłeś się stosować dziedziczenie szablonów, aby uprościć strukturę poszczególnych szablonów oraz ułatwić sobie zadanie modyfikowania witryny internetowej, gdy projekt będzie się rozrastał.

W rozdziale 19. utworzymy intuicyjne, przyjazne użytkownikom strony pozwalające im na dodawanie nowych tematów oraz wpisów, a także na edycję istniejących wpisów bez konieczności użycia witryny administracyjnej. Ponadto zajmiemy się budowaniem systemu rejestracji użytkowników, który pozwoli użytkownikom na tworzenie kont i wykorzystywanie naszej witryny internetowej. To jest serce aplikacji internetowej — możliwość utworzenia czegoś, z czego będzie korzystać niezliczona liczba użytkowników.

# 19

## Konta użytkowników



SERCEM APLIKACJI INTERNETOWEJ JEST UMOŻLIWIENIE KAŻDEMU UŻYTKOWNIKOWI ZNAJDUJĄCEMU SIĘ W DOWOLNYM MIEJSCU NA ŚWIECIE PRZEPROWADZENIA OPERACJI REJESTRACJI KONTA W APLIKACJI I ROzpoczęcia korzystania z niej. W tym rozdziale przygotujemy formularze pozwalające użytkownikom zakładać nowe tematy, dodawać nowe wpisy oraz edytować istniejące. Dowiesz się również, jak Django chroni przed najczęstszymi rodzajami ataków podejmowanych na strony oparte na formularzach. Dzięki temu nie będziesz musiał poświęcać zbyt wiele czasu na zabezpieczanie aplikacji.

Następnie przystąpimy do implementacji systemu uwierzytelniania użytkownika. Zbudujemy stronę rejestracji, która umożliwi użytkownikom założenie konta, oraz ograniczymy dostęp do wybranych stron, które będą mogli wyświetlić tylko zalogowani użytkownicy. Zmodyfikujemy niektóre funkcje widoku, aby użytkownicy mogli widzieć jedynie własne dane. Zobaczysz, jak można zapewnić bezpieczeństwo danym użytkowników.

### Umożliwienie użytkownikom wprowadzania danych

Zanim przejdziemy do budowania systemu uwierzytelniania przeznaczonego do tworzenia kont, zaczniemy od dodania pewnych stron pozwalających użytkownikom na wprowadzanie danych. Umożliwimy użytkownikom dodawanie nowych tematów i wpisów oraz edytowanie już istniejących wpisów.

Na razie tylko superużytkownik może wprowadzać dane za pomocą witryny administracyjnej. Ponieważ nie chcemy, aby użytkownicy korzystali z witryny administracyjnej, użyjemy oferowanych przez Django narzędzi do budowania formularzy, by w ten sposób przygotować strony pozwalające użytkownikom wprowadzać dane.

## Dodawanie nowego tematu

Rozpoczniemy od umożliwienia użytkownikom dodawania nowych tematów. Dodanie nowej strony opartej na formularzu odbywa się praktycznie w taki sam sposób, jaki stosowaliśmy dla wcześniej budowanych stron, czyli najpierw definiujemy adres URL, a później tworzymy funkcję widoku i szablon. Najważniejsza różnica polega na dodaniu nowego modułu o nazwie *forms.py*, zawierającego formularze.

## Formularz modelu dla tematu

Każda strona, która pozwala użytkownikowi na wprowadzanie i przekazywanie informacji na stronie internetowej, jest *formularzem*, nawet jeśli jej wygląd nie przypomina formularza. Kiedy użytkownik podaje informacje, konieczne jest przeprowadzenie ich *weryfikacji* i sprawdzenie, czy podane zostały odpowiedniego rodzaju dane, czy nie zawierają żadnego kodu o złośliwym działaniu, na przykład przeznaczonego do zakłócenia pracy serwera. Następnie podane informacje trzeba przetworzyć i zapisać w odpowiednim miejscu w bazie danych. Framework Django automatyzuje większość tej pracy.

Najprostszym sposobem utworzenia formularza w Django jest użycie tak zwanego *formularza modelu* (egzemplarz klasy *ModelForm*), który informacji pochodzących z modeli zdefiniowanych w rozdziale 18. używa do automatycznego utworzenia formularza. Pierwszy formularz umieszczamy w pliku *forms.py*, który należy utworzyć w tym samym katalogu co plik *models.py*.

Plik *forms.py*:

---

```
from django import forms

from .models import Topic

class TopicForm(forms.ModelForm): ❶
    class Meta:
        model = Topic ❷
        fields = ['text'] ❸
        labels = {'text': ''} ❹
```

---

Zaczynamy od zainportowania modułu *forms* i modelu, z którym będziemy pracować, czyli *Topic*. W wierszu ❶ definiujemy klasę o nazwie *TopicForm*, dziedziczącą po klasie *forms.ModelForm*.

Najprostsza wersja egzemplarza klasy ModelForm składa się z zagnieździonej klasy Meta, która wskazuje frameworkowi Django model będący podstawą dla formularza, oraz kolumn modelu, które mają być uwzględnione w budowanym formularzu. W wierszu ❷ budujemy formularz na podstawie modelu Topic i decydujemy się na uwzględnienie jedynie kolumny text (patrz wiersz ❸). Pusty ciąg tekstowy w słowniku labels (patrz wiersz ❹) informuje Django, że dla kolumny text nie powinna być generowana etykieta.

## Adres URL dla strony new\_topic

Adres URL dla nowej strony powinien być krótki i jasny. Dlatego też kiedy użytkownik będzie chciał dodać nowy temat, wówczas zostanie przekierowany na stronę [http://localhost:8000/new\\_topic/](http://localhost:8000/new_topic/). Poniżej przedstawiłem wzorzec adresu URL dla strony new\_topic, który należy umieścić w pliku learning\_logs/urls.py.

Plik learning\_logs/urls.py:

---

```
--cięcie--  
urlpatterns = [  
    --cięcie--  
    # Strona przeznaczona do dodawania nowego tematu.  
    path('new_topic/', views.new_topic, name='new_topic'),  
]
```

---

Ten wzorzec adresu URL będzie wykonywał żądania do funkcji widoku o nazwie new\_topic(), którą zdefiniujemy w następnej sekcji.

## Funkcja widoku new\_topic()

Funkcja widoku new\_topic() musi zajmować się obsługą dwóch różnych sytuacji. Pierwsza to początkowe żądania do strony new\_topic — w takim przypadku powinien zostać wyświetlony pusty formularz. Druga to przetworzenie wszelkich danych przekazanych w formularzu — w takim przypadku konieczne jest przekierowanie użytkownika z powrotem na stronę topics.

Plik views.py:

---

```
from django.shortcuts import render, redirect  
  
from .models import Topic  
from .forms import TopicForm  
  
--cięcie--  
def new_topic(request):  
    """Dodaj nowy temat."""  
    if request.method != 'POST': ❶  
        # Nie przekazano żadnych danych, należy utworzyć pusty formularz.  
        form = TopicForm() ❷  
    else:  
        pass
```

```
# Przekazano dane za pomocą żądania POST, należy je przetworzyć.  
form = TopicForm(data=request.POST) ❸  
if form.is_valid(): ❹  
    form.save() ❺  
    return redirect('learning_logs:topics') ❻  
  
# Wyświetlenie pustego formularza.  
context = {'form': form} ❼  
return render(request, 'learning_logs/new_topic.html', context)
```

---

Zimportowaliśmy funkcję `redirect()`, za pomocą której będziemy przekierowywać użytkownika, kiedy doda już nowy temat, z powrotem na stronę `topics`. Funkcja pobiera nazwę widoku i przekierowuje użytkownika do tego widoku. Zimportowaliśmy także przygotowany wcześniej formularz — `TopicForm`.

## Żądania GET i POST

Dwa podstawowe rodzaje żądań używanych podczas budowania aplikacji internetowych to GET i POST. Żądania GET są używane w przypadku stron, które jedynie odczytują dane z serwera. Natomiast z żądań POST korzystamy zwykle wtedy, gdy użytkownik musi przekazać informacje za pomocą formularza. Metodę POST będziemy wskazywać jako używaną do przetwarzania wszystkich budowanych tutaj formularzy. (Istnieje także kilka innych rodzajów żądań, ale nie będziemy z nich korzystać w omawianym projekcie).

Funkcja `new_topic()` pobiera obiekt żądania jako parametr. Kiedy użytkownik wykona początkowe żądanie strony, przeglądarka internetowa wykona żądanie GET. Natomiast gdy użytkownik wypełni formularz i wyśle go, wtedy przeglądarka internetowa wykona żądanie POST. W zależności od rodzaju żądania będziemy wiedzieli, czy użytkownik żąda wyświetlenia pustego formularza (żądanie GET) czy przetworzenia formularza już wypełnionego (żądanie POST).

Zdefiniowana w wierszu ❶ operacja sprawdzenia ma na celu ustalenie rodzaju użytej metody żądania (GET lub POST). Jeżeli metodą żądania nie jest POST, prawdopodobnie tą metodą jest GET, więc należy zwrócić pusty formularz. (W przypadku innego rodzaju żądania zwrot pustego formularza nadal można uznać za bezpieczne rozwiązanie). W wierszu ❷ tworzymy egzemplarz klasy `TopicForm`, przechowujemy go w zmiennej `form` i przekazujemy formularz do szablonu w słowniku kontekstu (patrz wiersz ❼). Ponieważ w trakcie tworzenia egzemplarza `TopicForm` nie podaliśmy żadnych argumentów, Django utworzy pusty formularz gotowy do wypełnienia przez użytkownika.

Jeżeli metodą żądania jest POST, nastąpi wykonanie bloku `else` i przetworzenie danych przekazanych w formularzu. Tworzymy egzemplarz klasy `TopicForm` (patrz wiersz ❸) i przekazujemy dane wprowadzone przez użytkownika, przechowywane w `request.POST`. Zwrócony obiekt `form` zawiera informacje, które zostały przekazane przez użytkownika.

Przekazanych informacji nie możemy zapisać w bazie danych aż do chwili sprawdzenia, czy na pewno są prawidłowe (patrz wiersz ❹). Funkcja `is_valid()`

sprawdza, czy wszystkie pola wymagane przez formularz zostały wypełnione (domyślnie wszystkie pola formularza są uznawane za obowiązkowe) oraz czy wprowadzone dane pasują do zdefiniowanego typu kolumny. Na przykład w kolumnie `text` powinno znajdować się mniej niż 200 znaków, jak to zdefiniowaliśmy w pliku `models.py` w poprzednim rozdziale. Tego rodzaju automatyczna weryfikacja oszczędza nam wiele pracy. Jeżeli wszystkie dane z formularza są prawidłowe, można wywołać metodę `save()`, która zapisuje je w bazie danych (patrz wiersz ⑤).

Po zapisaniu danych można już opuścić stronę. Funkcja `redirect()` pobiera nazwę widoku i przekierowuje użytkownika na stronę powiązaną z tym widokiem. W omawianym przykładzie funkcja `redirect()` służy do przekierowania użytkownika na stronę `topics` (patrz wiersz ⑥), na której powinien zobaczyć nowo dodany temat na liście wszystkich tematów.

Zmienna `context` została zdefiniowana na końcu funkcji widoku, a strona zostanie wygenerowana za pomocą szablonu `new_topic.html`, którego utworzeniem zajmiemy się za chwilę. Kod został umieszczony na zewnątrz wszystkich bloków `if`, więc zostanie wykonany w przypadku utworzenia pustego formularza lub po wysłaniu nieprawidłowego formularza. W tym drugim przypadku są dostępne domyślne komunikaty błędów, które mają pomóc użytkownikowi w podaniu prawidłowych danych w formularzu.

## Szablon dla strony `new_topic`

Przechodzimy teraz do utworzenia nowego szablonu o nazwie `new_topic.html` przeznaczonego do wyświetlenia zbudowanego przed chwilą formularza.

*Plik `new_topic.html`:*

---

```
{% extends "learning_logs/base.html" %}

{% block content %}
    <p>Dodaj nowy temat:</p>

    <form action="{% url 'learning_logs:new_topic' %}" method='post'> ❶
        {% csrf_token %} ❷
        {{ form.as_div }} ❸
        <button name="submit">Dodaj temat</button> ❹
    </form>

    {% endblock content %}
```

---

Ten szablon rozszerza `base.html`, aby strona zachowała podstawową strukturę, taką samą jak pozostałe strony w aplikacji *Learning Log*. W wierszu ❶ używamy znaczników `<form></form>` do zdefiniowania formularza HTML. Argument `action` wskazuje serwerowi miejsce docelowe dla danych przekazanych w formularzu. W omawianym przykładzie zostaną one przekierowane z powrotem do funkcji `new_topic()`. Argument `method` nakazuje przeglądarkce internetowej przekazanie danych jako żądania POST.

W wierszu ② Django używa szablonu znacznika `{% csrf_token %}`, aby uniemożliwić atakującemu użycie formularza do uzyskania nieautoryzowanego dostępu do serwera (tego rodzaju atak jest nazywany *cross-site request forgery*). W wierszu ③ wyświetlamy formularz. W tym miejscu możesz zobaczyć, jak Django ułatwia zadania takie jak wyświetlanie formularza. Konieczne jest jedynie podanie zmiennej formularza `{{ form.as_div }}`, a framework automatycznie utworzy wszystkie pola niezbędne do wyświetlenia tego formularza. Modyfikator `as_div` nakazuje Django wygenerowanie wszystkich elementów formularza w formie elementów HTML `<div></div>`, co jest prostym sposobem na eleganckie wyświetlenie formularza.

Django nie tworzy przycisku wysyłającego formularz, więc definiujemy go w wierszu ④.

## Dodanie łącza prowadzącego na stronę new\_topic

Następnym krokiem jest dodanie na stronie topics łącza prowadzącego na stronę `new_topic`.

Plik `topics.html`:

```
{% extends "learning_logs/base.html" %}

{% block content %}

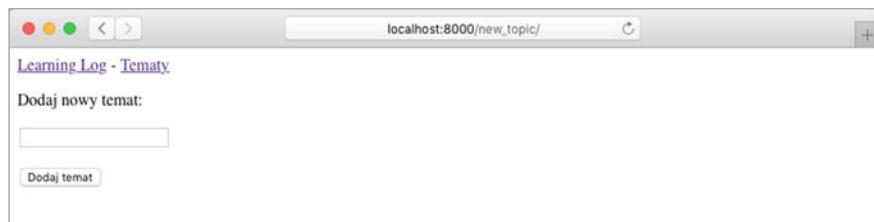
<p>Tematy</p>

<ul>
    --cięcie--
</ul>

<a href="{% url 'learning_logs:new_topic' %}">Dodaj nowy temat</a>

{% endblock content %}
```

Umieść łącze za listą istniejących tematów. Na rysunku 19.1 pokazałem przygotowany formularz. Możesz go teraz wykorzystać, aby dodać kilka nowych tematów.



Rysunek 19.1. Strona pozwalająca na dodanie nowego tematu

## Dodawanie nowych wpisów

Skoro użytkownik może już dodawać nowy temat, na pewno będzie chciał mieć również możliwość dodawania nowych wpisów. Ponownie zdefiniujemy adres URL, utworzymy funkcję widoku i szablon, a następnie łącze do strony. Jednak pracę musimy zacząć od dodania nowej klasy w pliku *forms.py*.

### Formularz modelu dla wpisu

Konieczne jest utworzenie formularza powiązanego z modelem *Entry*. Tym razem nieco bardziej zmodyfikujemy formularz niż w przypadku *TopicForm*.

Plik *forms.py*:

---

```
from django import forms

from .models import Topic, Entry

class TopicForm(forms.ModelForm):
    --cięcie--

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
        labels = {'text': ''} ❶
        widgets = {'text': forms.Textarea(attrs={'cols': 80})} ❷
```

---

Zaczynamy od uaktualnienia polecenia `import`, aby importowało nie tylko klasę *Topic*, ale również *Entry*. Nowa klasa *EntryForm* dziedziczy po `forms.ModelForm` i ma zagnieżdżoną klasę *Meta*. Klasa ta z kolei ma wskazany model, na którym bazuje, oraz określa kolumny przeznaczone do uwzględnienia w formularzu. Kolumnie 'text' ponownie przypisujemy pustą etykietę (patrz wiersz ❶).

W wierszu ❷ dodajemy atrybut `widgets`. Ten *widżet* to element formularza HTML, taki jak jednowierszowe pole tekstowe, wielowierszowe pole tekstowe lub rozwijana lista. Dzięki dołączeniu atrybutu `widget` można nadpisać domyślne widżety stosowane przez Django. Nakazując Django użycie elementu `forms.Textarea`, możemy dostosować do własnych potrzeb widżet danych wejściowych dla kolumny 'text', aby pole dla danych tekstowych miało szerokość 80 kolumn zamiast domyślnych 40. W ten sposób użytkownik będzie miał więcej miejsca podczas tworzenia nowego wpisu.

### Adres URL dla strony `new_entry`

W adresie URL przeznaczonym do dodawania nowego wpisu konieczne jest umieszczenie argumentu `topic_id`, ponieważ każdy wpis musi być powiązany z określonym tematem. Poniżej przedstawiłem wzorzec adresu URL, który trzeba dodać do *learning\_logs/urls.py*.

Plik learning\_logs/urls.py:

---

```
--cięcie--  
urlpatterns = [  
    --cięcie--  
    # Strona przeznaczona do dodawania nowego wpisu.  
    path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry'),  
]
```

---

Ten wzorzec adresu URL spowoduje dopasowanie każdego adresu URL w postaci `http://localhost:8000/new_entry/id/`, gdzie `id` oznacza liczbę dopasowującą identyfikator tematu. Kod `<int:topic_id>` powoduje przechwycenie wartości liczbowej i umieszczenie jej w zmiennej o nazwie `topic_id`. Kiedy żądany będzie adres URL dopasowany do tego wzorca, Django przekaże żądanie i identyfikator tematu do funkcji `new_entry()`.

## Funkcja widoku new\_entry()

Funkcja widoku dla strony `new_entry` jest w dużej mierze podobna do funkcji widoku odpowiedzialnej za obsługę dodawania nowego tematu. W pliku `views.py` umieść nowy kod.

Plik views.py:

---

```
from django.shortcuts import render, redirect  
  
from .models import Topic  
from .forms import TopicForm, EntryForm  
  
--cięcie--  
def new_entry(request, topic_id):  
    """Dodanie nowego wpisu dla określonego tematu."""  
    topic = Topic.objects.get(id=topic_id) ❶  
  
    if request.method != 'POST': ❷  
        # Nie przekazano żadnych danych, należy utworzyć pusty formularz.  
        form = EntryForm() ❸  
    else:  
        # Przekazano dane za pomocą żądania POST, należy je przetworzyć.  
        form = EntryForm(data=request.POST) ❹  
        if form.is_valid():  
            new_entry = form.save(commit=False) ❺  
            new_entry.topic = topic ❻  
            new_entry.save()  
            return redirect('learning_logs:topic', topic_id=topic_id) ❼  
  
    # Wyświetlenie pustego formularza.  
    context = {'topic': topic, 'form': form}  
    return render(request, 'learning_logs/new_entry.html', context)
```

---

Uaktualniamy polecenie `import`, aby obejmowało także utworzony wcześniej formularz `EntryForm`. Definicja funkcji `new_entry()` zawiera parametr `topic_id` przeznaczony do przechowywania wartości otrzymanej z adresu URL. Temat jest niezbędny do wygenerowania strony i przetworzenia danych formularza, więc wykorzystujemy `topic_id` do pobrania prawidłowego obiektu tematu w wierszu ❶.

W wierszu ❷ sprawdzamy rodzaj metody żądania (GET czy POST). W przypadku żądania GET następuje wykonanie bloku `if` i utworzenie egzemplarza pustego formularza `EntryForm` (patrz wiersz ❸).

Natomiast jeżeli metodą żądania jest POST, przetwarzamy dane przez utworzenie egzemplarza `EntryForm` zawierającego dane żądania POST pochodzące z obiektu `request` (patrz wiersz ❹). Następnie sprawdzamy, czy formularz jest prawidłowy. Jeżeli tak, konieczne jest zdefiniowanie atrybutu `topic` obiektu wpisu przed jego zapisaniem w bazie danych. Podczas wywoływania metody `save()` dodajemy argument `commit=False` (patrz wiersz ❺), aby nakazać Django utworzenie nowego obiektu wpisu i jego przechowywanie w zmiennej `new_entry`, ale jeszcze bez zapisywania w bazie danych. Atrybutowi `new_entry` egzemplarza `topic` przypisujemy temat pobrany z bazy danych na początku kodu funkcji (patrz wiersz ❻), a następnie wywołujemy `save()` bez argumentów. W ten sposób wpis zostanie zapisany w bazie danych wraz z przypisanym mu prawidłowym tematem.

Wywołanie `redirect()` w wierszu ❷ wymaga dwóch argumentów, mianowicie nazwy widoku, do którego ma zostać wykonane przekierowanie, oraz argumentu niezbędnego dla funkcji widoku. W omawianym przykładzie przekierowanie następuje do funkcji `topic()` wymagającej argumentu `topic_id`. Następnie ten widok powoduje wygenerowanie strony tematu, dla którego użytkownik utworzył nowy wpis. Na tej stronie użytkownik powinien zobaczyć nowy wpis na liście wszystkich wpisów dla danego tematu.

Na końcu funkcji następuje utworzenie słownika `context` i wygenerowanie strony na podstawie szablonu `new_entry.html`. Ten kod zostanie wykonany dla formularza pustego lub formularza wysłanego i uznanego za nieprawidłowy.

## Szablon dla strony new\_entry

Jak możesz zobaczyć w przedstawionym poniżej fragmencie kodu, szablon dla strony `new_entry` jest bardzo podobny do utworzonego wcześniej szablonu `new_topic`.

*Plik new\_entry.html:*

```
{% extends "learning_logs/base.html" %}

{% block content %}

<p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p> ❶

<p>Dodaj nowy wpis:</p>
<form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'> ❷
    {% csrf_token %}
```

```
    {{ form.as_div }}
    <button name='submit'>Dodaj wpis</button>
</form>

{% endblock content %}
```

---

Temat jest wyświetlany na początku strony (patrz wiersz ①), aby użytkownik nie miał żadnych wątpliwości, do jakiego tematu dodaje nowy wpis. Nazwa tematu działa również w charakterze łącza pozwalającego na przejście na stronę główną dla danego tematu.

Argument `action` formularza zawiera wartość `topic.id` pochodzązą z adresu URL, więc funkcja widoku może powiązać nowy wpis z odpowiednim tematem (patrz wiersz ②). Poza wymienionymi różnicami pozostała część szablonu jest taka sama jak w przypadku pliku `new_topic.html`.

## Dodanie łącza prowadzącego na stronę new\_entry

Następnym krokiem jest dodanie łącza prowadzącego na stronę `new_entry` i wyświetlanego na wszystkich stronach tematów.

*Plik topic.html:*

---

```
{% extends "learning_logs/base.html" %}

{% block content %}

<p>Temat: {{ topic }}</p>

<p>Wpisy:</p>
<p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Dodaj nowy wpis</a>
</p>

<ul>
--cięcie--
</ul>

{% endblock content %}
```

---

Łącze dodaliśmy tuż przed wyświetlaniem wpisów, ponieważ dodanie nowego wpisu będzie najczęściej podejmowaną akcją na tej stronie. Na rysunku 19.2 pokazalem stronę `new_entry` w działaniu. Teraz użytkownicy mogą dodawać nowe tematy i wpisy przeznaczone dla danych tematów. Wypróbuj stronę `new_entry` przez dodanie kilku wpisów do utworzonych wcześniej tematów.



Rysunek 19.2. Strona new\_entry

## Edycja wpisu

Przygotujemy teraz stronę pozwalającą użytkownikowi na edytowanie istniejącego wpisu.

### Adres URL dla strony edit\_entry

Adres URL dla strony edycji wpisu wymaga przekazania identyfikatora wpisu, który chcemy edytować. Poniżej pokazałem uaktualnioną wersję pliku *learning\_logs/urls.py*.

Plik urls.py:

```
--cięcie--  
urlpatterns = [  
    --cięcie--  
    # Strona przeznaczona do edycji wpisu.  
    path('edit_entry/<int:entry_id>/', views.edit_entry,  
         name='edit_entry'),  
]
```

Identyfikator przekazywany w adresie URL (na przykład *http://localhost:8000/edit\_entry/1/*) jest przechowywany w parametrze *entry\_id*. Żądanie, które jest dopasowane do tego formatu, wzorzec adresu URL przekazuje do funkcji widoku *edit\_entry()*.

### Funkcja widoku edit\_entry()

Kiedy funkcja *edit\_entry()* otrzyma żądanie GET, wówczas jej wartością zwrotną będzie formularz przeznaczony do edycji wpisu. Natomiast po otrzymaniu żądania POST, które będzie zawierać zmodyfikowany tekst wpisu, funkcja *edit\_entry()* zapisze ten tekst w bazie danych.

## Plik views.py:

---

```
from django.shortcuts import render, redirect  
  
from .models import Topic, Entry  
from .forms import TopicForm, EntryForm  
--cięcie--  
  
def edit_entry(request, entry_id):  
    """Edycja istniejącego wpisu."""  
    entry = Entry.objects.get(id=entry_id) ❶  
    topic = entry.topic  
  
    if request.method != 'POST':  
        # Żądanie początkowe, wypełnienie formularza aktualną treścią wpisu.  
        form = EntryForm(instance=entry) ❷  
    else:  
        # Przekazano dane za pomocą żądania POST, należy je przetworzyć.  
        form = EntryForm(instance=entry, data=request.POST) ❸  
        if form.is_valid():  
            form.save() ❹  
            return redirect('learning_logs:topic', topic_id=topic.id) ❺  
  
    context = {'entry': entry, 'topic': topic, 'form': form}  
    return render(request, 'learning_logs/edit_entry.html', context)
```

---

Na początku importujemy model `Entry`. W wierszu ❶ pobieramy obiekt wpisu, który użytkownik chce edytować, oraz temat powiązany z tym wpisem. W bloku `if` wykonywanym dla żądania GET tworzymy egzemplarz `EntryForm` wraz z argumentem `instance=entry` (patrz wiersz ❷). Ten argument nakazuje Django utworzenie formularza wypełnionego informacjami pochodząymi z obiektu istniejącego wpisu. Użytkownik zobaczy istniejące dane i będzie mógł je zmodyfikować.

Podczas przetwarzania żądania POST przekazujemy argumenty `instance=entry` i `data=request.POST` (patrz wiersz ❸) i tym samym nakazujemy Django utworzenie egzemplarza formularza na podstawie informacji powiązanych z obiektem istniejącego wpisu, uaktualnionego odpowiednimi danymi pochodzącymi z `request`. Następnie sprawdzamy, czy formularz jest prawidłowy. Jeżeli tak, wywołujemy metodę `save()` bez argumentów (patrz wiersz ❹). Nastepnym krokiem jest przekierowanie użytkownika na stronę `topic` (patrz wiersz ❺), na której powinien zobaczyć uaktualnioną wersję edytowanego wpisu.

Jeżeli wyświetlany jest formularz początkowy przeznaczony do edycji wpisu lub wysłany formularz został uznany za nieprawidłowy, wówczas następuje utworzenie słownika kontekstu i wygenerowanie strony za pomocą szablonu `edit_entry.html`.

## Szablon dla strony `edit_entry`

Poniżej przedstawiłem kod szablonu `edit_entry.html`, który jest bardzo podobny do szablonu `new_entry.html`.

### Plik edit\_entry.html:

---

```
{% extends "learning_logs/base.html" %}

{% block content %}

<p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

<p>Edycja wpisu:</p>

<form action="{% url 'learning_logs:edit_entry' entry.id %}" method="post"> ❶
    {% csrf_token %}
    {{ form.as_div }}
    <button name="submit">Zapisz zmiany</button> ❷
</form>

{% endblock content %}
```

---

W wierszu ❶ argument `action` powoduje przekazanie formularza z powrotem do funkcji `edit_entry()` w celu jego przetworzenia. Dołączamy identyfikator wpisu jako argument w znaczniku `{% url %}`, więc funkcja widoku może zmodyfikować odpowiedni obiekt wpisu. Przycisk wysyłający formularz ma etykietę `Zapisz zmiany`, aby przypomnieć użytkownikowi, że zapisuje zmiany dokonane w istniejącym wpisie, a nie tworzy nowy wpis (patrz wiersz ❷).

### Dodanie łącza prowadzącego na stronę edit\_entry

Następnym krokiem jest dodanie łącza prowadzącego na stronę `edit_entry` i wyświetlanego przy wszystkich wpisach na stronie tematu.

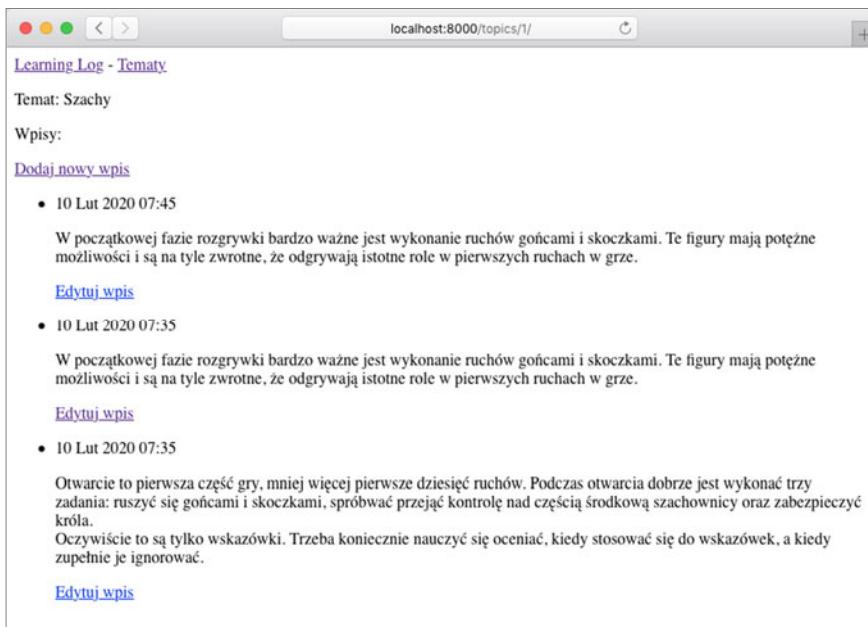
### Plik topic.html:

---

```
--cięcie--
{% for entry in entries %}
<li>
    <p>{{ entry.date_added|date:'d M Y H:i' }}</p>
    <p>{{ entry.text|linebreaks }}</p>
    <p>
        <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edytuj wpis</a>
    </p>
</li>
--cięcie--
```

---

Łącze pozwalające na edycję wpisu umieściliśmy po dacie i tekście danego wpisu. Znacznik szablonu `{% url %}` wykorzystaliśmy do określenia adresu URL dla wzorca adresu URL `edit_entry` wraz z atrybutem identyfikatora wskazującego w pętli aktualny wpis (`entry.id`). Tekst łącza `Edytuj wpis` jest wyświetlany po treści wpisu na stronie. Na rysunku 19.3 możesz zobaczyć stronę tematu zawierającą dodane łącza, które mają umożliwić edycję wpisów.



Rysunek 19.3. Każdy wpis ma teraz łącze pozwalające na edycję treści wpisu

Aplikacja *Learning Log* ma teraz większość funkcjonalności, które dla niej przewidzieliśmy. Użytkownicy mogą dodawać tematy i wpisy, a także czytać dowolne zestawy wpisów. W następnym podrozdziale przystąpimy do implementacji systemu rejestracji użytkownika, aby każdy mógł utworzyć konto w aplikacji *Learning Log* i tworzyć własne zestawy tematów oraz wpisów.

## ZRÓB TO SAM

**19.1. Blog.** Rozpocznij pracę nad nowym projektem Django o nazwie *Blog*. W tym projekcie utwórz aplikację o nazwie *blogs*, zdefiniuj w niej model przedstawiający ogólny blog oraz model przedstawiający post bloga. Każdy z modeli powinien mieć odpowiednie kolumny. Utwórz superużytkownika dla projektu, a następnie wykorzystaj witrynę administracyjną do przygotowania kilku krótkich postów. Strona główna powinna wyświetlać wszystkie posty w kolejności chronologicznej.

Przygotuj jeden formularz pozwalający na tworzenie nowych postów oraz drugi przeznaczony do edycji już istniejących. Wypełnij formularze i upewnij się, że działają prawidłowo.

# Konfiguracja kont użytkowników

W tym podrozdziale zajmiemy się opracowaniem systemu rejestracji kont użytkowników oraz ich autoryzacji, co pozwoli użytkownikom na rejestrację kont, a także logowanie się do aplikacji internetowej i wylogowanie się z niej. Utworzymy nową aplikację zawierającą całą funkcjonalność dotyczącą pracy z użytkownikami. W maksymalnym stopniu postaramy się wykorzystać domyślny system uwierzytelniania, który został wbudowany w Django. Ponadto nieco zmodyfikujemy model Topic, aby każdy temat należał do określonego użytkownika.

## Aplikacja accounts

Pracę rozpoczynamy od utworzenia nowej aplikacji o nazwie accounts. W tym celu wykorzystamy polecenie startapp:

```
(11_env)learning_log$ python manage.py startapp accounts
(11_env)learning_log$ ls
accounts db.sqlite3 learning_logs 11_env 11_project  manage.py ①
(11_env)learning_log$ ls accounts
__init__.py admin.py apps.py migrations models.py tests.py views.py ②
```

Domyślny system uwierzytelniania został opracowany na bazie koncepcji kont użytkowników (ang. *accounts*). Dlatego użycie nazwy *accounts* ułatwia integrację rozwiązania z domyślnym systemem. Wydanie polecenia startapp spowodowało utworzenie nowego katalogu o nazwie *accounts* (patrz wiersz ①) wraz ze strukturą identyczną jak w przypadku aplikacji *learning\_logs* (patrz wiersz ②).

## Dodanie aplikacji accounts do pliku settings.py

Konieczne jest dodanie naszej nowej aplikacji do listy INSTALLED\_APPS w pliku *settings.py*, jak pokazałem poniżej.

Plik *settings.py*:

```
--cięcie--
INSTALLED_APPS = [
    # Moje aplikacje.
    'learning_logs',
    'accounts',

    # Domyślne aplikacje Django.
--cięcie--
]
```

Teraz Django dołączy aplikację *accounts* do całego projektu.

## Dołączanie adresów URL z aplikacji accounts

Następnym krokiem jest modyfikacja głównego pliku *urls.py*, aby uwzględniać adresy URL, które przygotujemy dla aplikacji accounts.

*Plik ll\_project/urls.py:*

---

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('', include('learning_logs.urls')),
]
```

---

W powyższym fragmencie kodu dodaliśmy wiersz dołączający plik *urls.py* z aplikacji accounts. Ten wiersz spowoduje dopasowanie każdego adresu URL rozpoczynającego się od słowa *accounts*, na przykład *http://localhost:8000/accounts/login/*.

## Strona logowania

Najpierw zajmiemy się implementacją strony logowania. Wykorzystamy domyślny widok *login* wbudowany w Django, więc wzorzec adresów URL będzie wyglądał nieco inaczej niż wcześniejsze wzorce. Utwórz nowy plik *urls.py* w katalogu *ll\_project/accounts/* i umieść w nim poniższy fragment kodu.

*Plik accounts/urls.py:*

---

```
"""Definiuje wzorce adresów URL dla aplikacji accounts."""

from django.urls import path, include

app_name = 'accounts'
urlpatterns = [
    # Dołączanie domyślnych adresów URL uwierzytelniania.
    path('', include('django.contrib.auth.urls')),
]
```

---

Na początku importujemy funkcję *path()*, a następnie funkcję *include()*, aby można było dołączać zdefiniowane przez Django domyślne adresy URL uwierzytelniania. Te adresy domyślne zawierają wzorce takie jak '*login*' i '*logout*'. Zmiennej *app\_name* przypisujemy wartość '*accounts*', aby umożliwić Django odróżnianie wspomnianych adresów URL od należących do innych aplikacji projektu. Domyślne adresy URL dostarczane przez Django, po dołączeniu także tych zdefiniowanych w pliku *urls.py* aplikacji *accounts*, będą dostępne poprzez przestrzeń nazw *accounts*.

Wzorzec strony logowania powoduje dopasowanie adresu URL w postaci `http://localhost:8000/accounts/login/`. Kiedy Django odczyta ten adres URL, słowo `accounts` nakaże sprawdzenie pliku `accounts/urls.py`, natomiast słowo `login` nakaże przekazywanie żądań do wbudowanego w Django widoku `login`.

## Szablon dla strony logowania

Kiedy użytkownik zażąda wyświetlenia strony logowania, domyślnie Django wykorzysta wbudowany widok `login`, choć nadal konieczne jest dostarczenie szablonu dla tej strony. Domyślne widoki uwierzytelniania szukają szablonów w katalogu o nazwie `registration`, co oznacza konieczność utworzenia tego katalogu. W katalogu `ll_project/accounts/` utwórz podkatalog o nazwie `templates`, a wewnątrz niego podkatalog o nazwie `registration`. Poniżej przedstawiłem kod szablonu `login.html`, który powinien znajdować się w katalogu `ll_project/accounts/templates/registration/`.

Plik `login.html`:

```
{% extends "learning_logs/base.html" %}

{% block content %}

{%- if form.errors %} ❶
<p>Nazwa użytkownika i hasło są nieprawidłowe. Proszę spróbować ponownie.</p>
{%- endif %}

<form method="post" action="{% url 'accounts:login' %}"> ❷
{%- csrf_token %}
{{ form.as_div }} ❸

<button name="submit">Zaloguj</button> ❹
</form>

{% endblock content %}
```

Ten szablon rozszerza `base.html`, co gwarantuje, że strona logowania będzie miała taki sam wygląd i sposób działania jak pozostała część witryny internetowej. Zwróć uwagę na możliwość rozszerzenia w aplikacji szablonu pochodzącego z innej aplikacji.

Jeżeli będzie istniał atrybut `errors` formularza, wyświetlimy komunikat błędu (patrz wiersz ❶) informujący, że podana nazwa użytkownika i hasło nie pasują do informacji przechowywanych w bazie danych.

Widok logowania ma przetworzyć formularz, więc wartością argumentu `action` jest adres URL strony logowania (patrz wiersz ❷). Kiedy formularz zostanie przekazany do szablonu, programista stanie się odpowiedzialny za wyświetlenie tego formularza (patrz wiersz ❸) i dodanie przycisku go wysyłającego (patrz wiersz ❹).

## Ustawienie LOGIN\_REDIRECT

Gdy użytkownik zostanie zalogowany, Django musi wiedzieć, na jaką stronę go przekierować. Można to kontrolować za pomocą odpowiedniej opcji w pliku ustawień.

Na końcu pliku *settings.py* umieść następujący kod:

*Plik settings.py:*

```
--cięcie--  
# Moje ustawienia.  
LOGIN_REDIRECT_URL = 'learning_logs:index'
```

W pliku *settings.py* znajduje się wiele ustawień domyślnych, więc dobrze jest oddzielić sekcję przeznaczoną dla ustawień niestandardowych. Pierwszym dodanym w nowej sekcji jest `LOGIN_REDIRECT_URL`, wskazujące Django adres URL, do którego ma nastąpić przekierowanie po zakończonej powodzeniem operacji logowania.

## Dodanie łącza prowadzącego na stronę logowania

Łącze prowadzące na stronę logowania umieścimy w szablonie `base.html`, aby było wyświetlane na każdej stronie. Ponieważ nie chcemy wyświetlania tego łącza, gdy użytkownik jest zalogowany, umieszczamy je wewnątrz znacznika `{% if %}`.

*Plik base.html:*

```
<p>  
    <a href="{% url 'learning_logs:index' %}">Learning Log</a> -  
    <a href="{% url 'learning_logs:topics' %}">Tematy</a> -  
    {% if user.is_authenticated %} ❶  
        Witaj, {{ user.username }}. ❷  
    {% else %}  
        <a href="{% url 'accounts:login' %}">Zaloguj</a> ❸  
    {% endif %}  
</p>  
  
{% block content %}{% endblock content %}
```

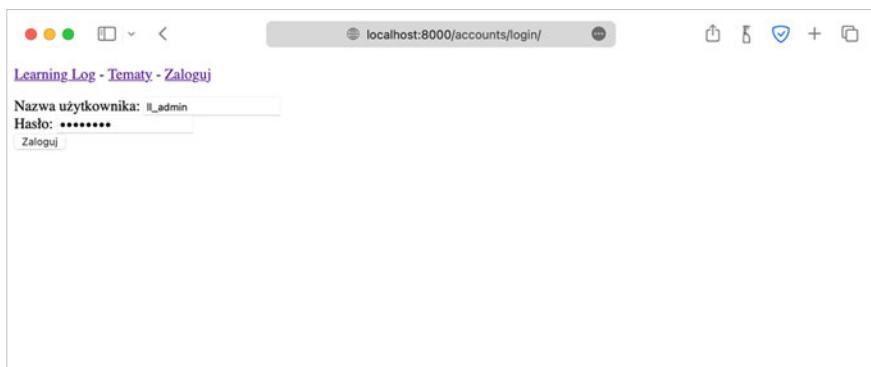
W systemie uwierzytelniania Django każdy szablon ma obiekt `user`, który zawsze ma przypisany atrybut `is_authenticated`. Wartością tego atrybutu jest `True`, gdy użytkownik będzie zalogowany i `False` w przeciwnym przypadku. W ten sposób wyświetlany komunikat można zróżnicować w zależności od tego, czy użytkownik jest uwierzytelniony.

W omawianym przykładzie wyświetlamy komunikat powitania użytkownikowi, który jest aktualnie zalogowany (patrz wiersz ❶). Uwierzytelniony użytkownik ma przypisany dodatkowy atrybut `username`, który wykorzystamy do personalizowania komunikatu powitania i przypomnienia użytkownikowi, że jest zalogowany (patrz wiersz ❷). W wierszu ❸ wyświetlamy łącze do strony logowania, ale tylko nieuwierzytelnionym użytkownikom.

## Użycie strony logowania

Skoro przygotowaliśmy konto użytkownika, spróbujmy się zalogować, aby zobaczyć, czy strona działa. Przejdz pod adres <http://localhost:8000/admin/>. Jeżeli nadal jesteś zalogowany jako admin, odszukaj w nagłówku łącze wylogowania i kliknij je.

Po wylogowaniu przejdź pod adres <http://localhost:8000/accounts/login/>. Powinieneś zobaczyć stronę logowania podobną do pokazanej na rysunku 19.4. Po daj zdefiniowane wcześniej dane uwierzytelniające, a po zakończonym sukcesem logowaniu zostaniesz przeniesiony z powrotem na stronę główną. Nagłówek na stronie głównej powinien wyświetlać spersonalizowany komunikat powitania zawierającego nazwę użytkownika.



Rysunek 19.4. Strona logowania

## Wylogowanie

Konieczne jest umożliwienie użytkownikom wylogowania się z aplikacji. Żądanie wylogowania powinno zostać wykonane jako żądanie POST. Dlatego w szablonie *base.html* zdefiniujemy łącze wylogowania, po którego kliknięciu użytkownik zostanie przeniesiony na stronę potwierdzającą wylogowanie.

### Dodanie do szablonu *base.html* formularza wylogowania

Formularz umożliwiający wylogowanie się użytkownika umieszczamy w szablonie *base.html*, więc będzie dostępny na każdej stronie. Definiujemy go w innym bloku `if`, więc będzie wyświetlany jedynie uwierzytelionym użytkownikom.

Plik *base.html*:

```
--cięcie--  
{% block content %}{% endblock content %}  
  
{% if user.is_authenticated %}  
  <hr /> ❶  
  <form action="{% url 'accounts:logout' %}" method='post'> ❷
```

```
{% csrf_token %}  
<button name='submit'>Wyloguj</button>  
</form>  
{% endif %}
```

---

Domyślnym wzorcem URL dla wylogowania jest 'accounts/logout/'. Jednak to żądanie musi być wykonane jako żądanie POST, w przeciwnym razie atakujący może łatwo wymuszać wykonywanie żądań wylogowania. Aby proces wylogowania używał żądań POST, definiujemy prosty formularz.

Formularz zostaje umieszczony u dołu strony, pod elementem `<hr />` wyświetlającym linię poziomą ①. Dzięki temu przycisk wylogowania zawsze będzie się znajdował w tym samym miejscu na każdej stronie. W formularzu wartością argumentu `action` jest adres URL wylogowania, natomiast jako metodę żądania wskażaliśmy 'post' ②. Każdy formularz w Django, nawet tak prosty jak w omawianym przykładzie, musi mieć zdefiniowany element `{% csrf_token %}`. Ten formularz jest praktycznie pusty i zawiera tylko wysyłający go przycisk.

## Ustawienie LOGOUT\_REDIRECT

Gdy użytkownik kliknie przycisk wylogowania, Django musi wiedzieć, na jaką stronę go przekierować. Można to kontrolować za pomocą odpowiedniej opcji w pliku ustawień `settings.py`.

Plik `settings.py`:

---

```
--cięcie--  
# Moje ustawienia.  
LOGIN_REDIRECT_URL = 'learning_logs:index'  
LOGOUT_REDIRECT_URL = 'learning_logs:index'
```

---

Ustawienie `LOGIN_REDIRECT_URL` nakazuje Django przekierowanie wylogowanego użytkownika na stronę główną. Jest to prosty sposób potwierdzenia operacji wylogowania, ponieważ wtedy użytkownik nie będzie już widział wyświetlonej swojej nazwy użytkownika.

## Strona rejestracji użytkownika

Następnym krokiem jest zbudowanie strony pozwalającej nowym użytkownikom na dokonanie rejestracji. Wprawdzie wykorzystamy wbudowany w Django formularz `UserCreationForm`, ale utworzymy własną funkcję widoku i szablon.

## Adres URL dla strony rejestracji użytkownika

Poniższy fragment kodu przedstawia wzorzec adresu URL dopasowującego stronę pozwalającą użytkownikowi na dokonanie rejestracji. Także i ten wzorzec należy umieścić w pliku `accounts/urls.py`.

## Plik accounts/urls.py:

---

```
"""Definiuje wzorce URL dla użytkowników."""

from django.urls import path, include

from . import views

app_name = 'accounts'
urlpatterns = [
    # Doliczenie domyslnych adresow URL uwierzytelniania.
    path('', include('django.contrib.auth.urls')),
    # Strona rejestracji.
    url('register/', views.register, name='register'),
]
```

---

Importujemy moduł `views` z aplikacji `accounts`, co jest konieczne, ponieważ tworzymy własny widok dla strony rejestracji. Ten wzorzec spowoduje dopasowanie adresu URL w postaci `http://localhost:8000/accounts/register/` i przekaże żądania do funkcji `register()`, którą wkrótce utworzymy.

## Funkcja widoku register()

Gdy żądanie zostanie wysłane po raz pierwszy, funkcja widoku `register()` powinna wyświetlić pusty formularz rejestracji, a następnie przetworzyć go, jak zostanie już wypełniony. Kiedy rejestracja zakończy się powodzeniem, funkcja powinna również zalogować nowego użytkownika. Umieść poniższy fragment kodu w pliku `accounts/views.py`.

## Plik accounts/views.py:

---

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Rejestracja nowego użytkownika."""
    if request.method != 'POST':
        # Wyświetlenie pustego formularza rejestracji użytkownika.
        form = UserCreationForm() ①
    else:
        # Przetworzenie wypełnionego formularza.
        form = UserCreationForm(data=request.POST) ②

        if form.is_valid(): ③
            new_user = form.save() ④
            # Zalogowanie użytkownika, a następnie przekierowanie go na stronę główną.
            login(request, new_user) ⑤
            return redirect('learning_logs:index') ⑥
```

```
# Wyświetlenie pustego formularza.  
context = {'form': form}  
return render(request, 'registration/register.html', context)
```

---

Zaczynamy od zimportowania funkcji `render()` i `register()`. Następnie importujemy funkcję `login()` pozwalającą użytkownikowi na zalogowanie się po podaniu poprawnych danych uwierzytelniających. Importujemy także domyślny formularz `UserCreationForm`. W funkcji `register()` sprawdzamy, czy udzielana jest odpowiedź na żądanie POST. Jeżeli nie, to tworzymy egzemplarz `UserCreationForm` bez żadnych danych początkowych (patrz wiersz ❶).

Jeżeli udzielana jest odpowiedź na żądanie POST, tworzymy egzemplarz `UserCreationForm` na podstawie przekazanych danych (patrz wiersz ❷). Sprawdzamy prawidłowość danych (patrz wiersz ❸) — w omawianym przypadku to będzie sprawdzenie, czy nazwa użytkownika ma odpowiednie znaki, czy hasło zostało dopasowane oraz czy użytkownik nie próbuje w przekazanych danych wprowadzić jakiegokolwiek kodu o złośliwym działaniu.

Jeżeli przekazane dane są prawidłowe, wywołujemy metodę `save()` w celu zapisania w bazie danych nazwy użytkownika i wartości hash wygenerowanej na podstawie hasła (patrz wiersz ❹). Metoda `save()` zwraca nowo utworzony obiekt użytkownika, który przechowujemy w zmiennej `new_user`. Po zapisaniu informacji przeprowadzamy logowanie użytkownika. Odbywa się to za pomocą wywołania funkcji `login()` wraz z obiektami `request` i `new_user` (patrz wiersz ❺), która powoduje utworzenie prawidłowej sesji dla nowego użytkownika. Na końcu przerzbowujemy użytkownika na stronę główną (patrz wiersz ❻), na której spersonalizowane powitanie w nagłówku informuje o zakończonej sukcesem rejestracji.

Na końcu funkcji mamy polecenie odpowiedzialne za wygenerowanie strony, która będzie zawierała pusty formularz lub wysłany formularz uznany za nieprawidłowy.

## Szablon dla strony rejestracji użytkownika

Szablon przeznaczony dla strony rejestracji użytkownika jest niezwykle podobny do używanego na stronie logowania. Upewnij się, że zapisales go w tym samym katalogu, w którym znajduje się już plik `login.html`.

*Plik register.html:*

---

```
{% extends "learning_logs/base.html" %}  
  
{% block content %}  
  
    <form method="post" action="{% url 'accounts:register' %}">  
        {% csrf_token %}  
        {{ form.as_div }}  
  
        <button name="submit">Rejestruj</button>  
    </form>  
  
{% endblock content %}
```

---

Ten szablon wygląda podobnie do innych szablonów opartych na formularzu, które zostały wcześniej utworzone. Ponownie wykorzystujemy metodę `as_div`, aby framework Django prawidłowo wyświetlał wszystkie pola formularza, w tym także wszelkie komunikaty błędów, które zostaną wygenerowane, jeśli formularz nie będzie poprawnie wypełniony.

## Dodanie łącza prowadzącego na stronę rejestracji użytkownika

Następnym krokiem jest dodanie kodu, który każdemu niezalogowanemu użytkownikowi wyświetli łącze prowadzące na stronę rejestracji.

*Plik base.html:*

---

```
--cięcie--  
    {%- if user.is_authenticated %}  
        Witaj, {{ user.username }}.  
        <a href="{% url 'accounts:logout' %}">Wyloguj</a>  
    {% else %}  
        <a href="{% url 'accounts:register' %}">Rejestruj</a> -  
        <a href="{% url 'accounts:login' %}">Zaloguj</a>  
    {% endif %}  
--cięcie--
```

---

Teraz zalogowany użytkownik będzie widział spersonalizowany komunikat powitania oraz łącze pozwalające się wylogować. Z kolei niezalogowany użytkownik będzie miał wyświetlane łącza pozwalające zarejestrować się i zalogować. Wypróbuj stronę rejestracji, tworząc kilka kont użytkowników o różnych nazwach.

W kolejnym podrozdziale zajmiemy się ograniczeniem dostępu do wybranych stron — będą one mogły być wyświetlane tylko przez zarejestrowanych użytkowników. Zagwarantujemy także, że każdy temat będzie przypisany do konkretnego użytkownika.

**UWAGA** Przygotowany tutaj system rejestracji pozwala na utworzenie dowolnej liczby kont użytkowników w Learning Log. Jednak część systemów wysyła użytkownikom wiadomość e-mail, na którą dany użytkownik musi odpowiedzieć, aby w ten sposób potwierdzić swoją tożsamość. Te systemy w porównaniu do prostych systemów, takich jak system zbudowany w tym rozdziale, tworzą mniej kont używanych przez spamerów. Jednak kiedy dopiero poznajesz sposoby budowania aplikacji, całkowicie uzasadnione jest eksperymentowanie z użyciem prostego systemu rejestracji użytkowników, takiego właśnie jak ten przedstawiony w tym rozdziale.

## ZRÓB TO SAM

**19.2. Konta bloga.** Do projektu bloga zapoczątkowanego w ćwiczeniu 19.1 dodaj systemu rejestracji i uwierzytelniania użytkowników. Upewnij się, że zalogowany użytkownik ma wyświetlzoną na ekranie swoją nazwę użytkownika, natomiast niezarejestrowany otrzymuje dostęp do strony pozwalającej mu na dokonanie rejestracji.

# Umożliwienie użytkownikom bycia właścicielami swoich danych

Użytkownik powinien mieć możliwość tworzenia danych, do których tylko on będzie miał dostęp. Dlatego też opracujemy system ustalający, kto jest właścicielem poszczególnych danych, a następnie ograniczymy dostęp do wybranych stron, aby użytkownik mógł pracować jedynie z własnymi danymi.

W tym podrozdziale zmodyfikujemy model `Topic` tak, aby każdy temat należał do konkretnego użytkownika. To przeniesie się również na wpisy, ponieważ każdy wpis należy do określonego tematu. Pracę rozpoczynamy od ograniczenia dostępu do wybranych stron.

## Ograniczenie dostępu za pomocą dekoratora `@login_required`

Za pomocą dekoratora `@login_required` Django bardzo ułatwia ograniczenie dostępu do wybranych stron, tak aby mogły być wyświetlane jedynie przez zalogowanych użytkowników. Przypomnij sobie z rozdziału 11., że *dekorator* to dyrektywa umieszczona przed definicją funkcji i modyfikująca sposób jej działania. Spójrz na przedstawiony poniżej przykład.

## Ograniczenie dostępu do strony tematów

Każdy temat będzie miał swojego właściciela, więc tylko zarejestrowani użytkownicy powinni mieć możliwość wyświetlania strony tematów. Umieść poniższy fragment kodu w pliku `learning_logs/views.py`.

*Plik `learning_logs/views.py`:*

---

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--cięcie--
```

```
@login_required  
def topics(request):  
    """Wyświetlenie wszystkich tematów."""  
    --cięcie--
```

---

Na początku importujemy funkcję `login_required()`. Tę funkcję stosujemy jako dekoratora dla funkcji widoku `topics()`, co odbiera się przez poprzedzenie `login_required` znakiem @, aby Python wiedział o konieczności wykonania kodu zdefiniowanego w funkcji `login_required()` przed przystąpieniem do wykonywania kodu funkcji `topics()`.

Kod funkcji `login_required()` sprawdza, czy użytkownik jest zalogowany, ponieważ to jest warunkiem koniecznym do wykonania funkcji `topics()` przez Django. Jeżeli użytkownik nie jest zalogowany, zostanie przekierowany na stronę logowania.

Aby przedstawione rozwiązanie działało, trzeba zmodyfikować plik `settings.py` i wskazać Django miejsce, w którym znajduje się strona logowania. Poniższy fragment kodu umieść na końcu pliku `settings.py`.

*Plik settings.py:*

---

```
--cięcie--  
# Moje ustawienia.  
LOGIN_REDIRECT_URL = 'learning_logs:index'  
LOGOUT_REDIRECT_URL = 'learning_logs:index'  
LOGIN_URL = 'accounts:login'
```

---

Teraz, kiedy nieuwierzytelny użytkownik wykona żądanie do strony chronionej przez dekoratora `@login_required`, Django przekieruje użytkownika do adresu URL zdefiniowanego przez `LOGIN_URL` w pliku `settings.py`.

Przygotowane rozwiązanie możesz przetestować, wylogowując się ze wszystkich kont użytkowników i przechodząc na stronę główną. Następnie kliknij łącze *Tematy*, które powinno przekierować Cię na stronę logowania. Teraz zaloguj się do jednego z dostępnych kont użytkowników i na stronie głównej ponownie kliknij łącze *Tematy*. W przeglądarce internetowej powinieneś zobaczyć wyświetlana stronę tematów.

## Ograniczenie dostępu w aplikacji Learning Log

Django bardzo ułatwia ograniczanie dostępu do stron, ale należy podjąć decyzję, które strony będą chronione. Znacznie lepsze podejście polega na ustaleniu stron niewymagających ochrony, a następnie ograniczeniu dostępu do wszystkich pozostałych w projekcie. Niepotrzebnie chronioną stronę zawsze można udostępnić — to znacznie mniej niebezpieczne niż pozostawienie niezabezpieczonych stron zawierających informacje wrażliwe.

W aplikacji *Learning Log* pozostawimy jako publicznie dostępne strony główną, rejestracji i wylogowania. Wszystkie pozostałe strony będą chronione.

Poniżej przedstawiłem plik *learning\_logs/views.py* wraz z dekoratorami `@login_required` zastosowanymi dla każdego widoku poza `index()`.

*Plik learning\_logs/views.py:*

---

```
--cięcie--  
@login_required  
def topics(request):  
    --cięcie--  
  
@login_required  
def topic(request, topic_id):  
    --cięcie--  
  
@login_required  
def new_topic(request):  
    --cięcie--  
  
@login_required  
def new_entry(request, topic_id):  
    --cięcie--  
  
@login_required  
def edit_entry(request, entry_id):  
    --cięcie--
```

---

Jeśli spróbujesz uzyskać dostęp do którejkolwiek z tych stron jako niezalogowany użytkownik, zostaniesz przekierowany na stronę logowania. Ponadto nie będziesz mieć możliwości klikania łączy do stron takich jak `new_topic`. Jeżeli zaś wprowadzisz adres URL w postaci `http://localhost:8000/new_topic/`, zostaniesz przekierowany na stronę logowania. Powinieneś ograniczyć dostęp do wszystkich adresów URL, które są dostępne publicznie i powiązane z prywatnymi danymi użytkowników.

## Powiązanie danych z określonymi użytkownikami

Musimy teraz powiązać przekazane dane z użytkownikiem, który je przekazał. Konieczne jest powiązanie z użytkownikiem jedynie danych znajdujących się najwyżej w hierarchii, ponieważ dane znajdujące się na niższych poziomach będą wówczas wiązane automatycznie. Na przykład w aplikacji *Learning Log* tematy są zaliczane do danych najwyższego poziomu, a wszystkie wpisy są powiązane z określonymi tematami. Dopóki każdy temat należy do konkretnego użytkownika, dopóty będziesz mieć możliwość monitorowania własności poszczególnych wpisów w bazie danych.

Zmodyfikujemy model `Topic` przez dodanie do użytkownika powiązania z klawiszem zewnętrznym. To będzie oznaczało konieczność przeprowadzenia migracji. Na końcu zmodyfikujemy widoki tak, aby wyświetlały jedynie dane należące do aktualnie zalogowanego użytkownika.

## Modyfikacja modelu Topic

W pliku *models.py* zmieniamy jedynie dwa wiersze kodu.

Plik *models.py*:

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """Temat poznawany przez użytkownika."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        """Zwraca reprezentację modelu w postaci ciągu tekstowego."""
        return self.text

class Entry(models.Model):
    --cięcie--
```

Zaczynamy od zimportowania modelu *User* z *django.contrib.auth*. Następnym krokiem jest dodanie kolumny *owner* do modelu *Topic*, co powoduje utworzenie związku klucza zewnętrznego z modelem *User*. Po usunięciu użytkownika wszystkie powiązane z nim tematy również zostaną usunięte.

## Identyfikacja istniejących użytkowników

Kiedy będzie wykonywana migracja bazy danych, Django zmodyfikuje bazę danych w taki sposób, aby mogła przechowywać informacje o połączeniu między każdym tematem i użytkownikiem. Aby przeprowadzić migrację, Django musi wiedzieć, którego użytkownika powiązać z istniejącymi tematami. Najprostsze podejście polega na przypisaniu wszystkich istniejących tematów jednemu użytkownikowi, na przykład superużytkownikowi. Przede wszystkim konieczne jest ustalenie identyfikatora tego użytkownika.

Spójrzmy na identyfikatory wszystkich utworzonych dotąd użytkowników. Uruchom sesję Django w powłoce, a następnie wydaj poniższe polecenia:

```
(11_env)learning_log$ python manage.py shell
>>> from django.contrib.auth.models import User ❶
>>> User.objects.all() ❷
<QuerySet [<User: ll_admin>, <User: eric>, <User: willie>]>
>>> for user in User.objects.all(): ❸
...     print(user.username, user.id)
...
ll_admin 1
eric 2
willie 3
>>>
```

W wierszu ❶ importujemy model `User` do sesji w powloce. Następnie analizujemy wszystkich utworzonych dotąd użytkowników (patrz wiersz ❷). Wygenerowane dane wyjściowe wskazują na istnienie trzech użytkowników: `ll_admin`, `eric` i `willie`.

W wierszu ❸ przeprowadzamy iterację przez listę wszystkich użytkowników, wyświetlając przy tym nazwę każdego z nich oraz przypisany im identyfikator. Kiedy Django zapyta o użytkownika, którego należy powiązać z istniejącymi tematami, wtedy podamy jeden z wyświetlonych tutaj identyfikatorów.

## Migracja bazy danych

Skoro znamy identyfikatory użytkowników, możemy przystąpić do migracji bazy danych. W jej trakcie Python zaproponuje tymczasowe połączenie modelu `Topic` z określonym użytkownikiem lub dodanie wartości domyślnej do pliku `models.py` – wybór należy do Ciebie. W omawianym projekcie wybierz pierwszą opcję.

---

```
(11_env)learning_log$ python manage.py makemigrations learning_logs ❶
It is impossible to add a non-nullable field 'owner' to topic without ❷
specifying a default. This is because...
Please select a fix: ❸
1) Provide a one-off default now (will be set on all existing rows with a
   null value for this column)
2) Quit and manually define a default value in models.py.
Select an option: 1 ❹
Please enter the default value now, as valid Python ❺
The datetime and django.utils.timezone modules are available...
Type 'exit' to exit this prompt
>>> 1 ❻
Migrations for 'learning_logs':
    learning_logs/migrations/0003_topic_owner.py:
        - Add field owner to topic
(11_env)learning_log$
```

---

Rozpoczynamy od wydania polecenia `makemigrations` w wierszu ❶. W wygenerowanych danych wyjściowych (patrz wiersz ❷) Django wskazuje, że próbujemy dodać wymaganą (wartość inna niż `NULL`) kolumnę do istniejącego modelu (`topic`) bez zdefiniowania wartości domyślnej. Framework daje nam do wyboru dwie możliwości rozwiązania tego problemu (patrz wiersz ❸). Pierwsza to podanie w tym momencie wartości domyślnej, natomiast druga to zakończenie operacji i dodanie wartości domyślnej w pliku `models.py`. W wierszu ❹ decydujemy się na pierwszą opcję. Django prosi więc o podanie wartości domyślnej (patrz wiersz ❺).

Aby powiązać wszystkie istniejące tematy z pierwotnym użytkownikiem `ll_admin`, w wierszu ❻ podałem identyfikator użytkownika 1. Możesz w tym miejscu podać identyfikator dowolnego utworzonego użytkownika, to naprawdę nie musi być superużytkownik. Następnie Django przeprowadza migrację bazy danych

z użyciem podanej wartości i generuje plik migracji `0003_topic_owner.py`, który dodaje kolumnę `owner` do modelu `Topic`.

Teraz można już przeprowadzić migrację. W aktywnym środowisku wirtualnym wydaj poniższe polecenie:

```
(11_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
  Applying learning_logs.0003_topic_owner... OK ❶
(11_env)learning_log$
```

Django przeprowadza nową migrację, a jej wynikiem jest OK (patrz wiersz ❶).

Za pomocą sesji Django w powłoce, takiej jak pokazałem poniżej, można sprawdzić, czy migracja zadziałała zgodnie z oczekiwaniami:

```
>>> from learning_logs.models import Topic
>>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Szachy 11_admin
Wspinaczka górska 11_admin
>>>
```

Importujemy model `Topic` z `learning_logs.models`, a następnie przeprowadzamy iterację przez wszystkie istniejące tematy, wyświetlając przy tym nazwę każdego z nich oraz nazwę użytkownika będącego właścicielem danego tematu. Jak widzisz, wszystkie tematy należą teraz do użytkownika `11_admin`. (Jeżeli wykonanie tego kodu zakończy się błędem, spróbuj zamknąć powłokę, a następnie ponownie uruchom nową).

**UWAGA** *Zamiast przeprowadzać migracji, bazę danych możesz po prostu wyzerować, co jednak oznacza utratę wszystkich istniejących danych. Dobrą praktyką jest przeprowadzanie migracji bazy danych przy jednoczesnym zachowaniu spójności danych użytkowników. Jeżeli chcesz rozpocząć pracę z nową bazą danych, wydaj polecenie `python manage.py flush`, które spowoduje odbudowę struktury bazy danych. Konieczne będzie ponowne utworzenie superużytkownika oraz odtworzenie wszystkich danych.*

## Przyznanie dostępu jedynie odpowiednim użytkownikom

Obecnie każdy zalogowany użytkownik będzie miał dostęp do wszystkich tematów niezależnie od tego, czy zostały utworzone przez niego. Wprowadzimy teraz zmianę polegającą na tym, że użytkownik będzie mógł wyświetlać jedynie tematy utworzone przez siebie.

Wprowadź następującą zmianę w funkcji `topic()` w pliku `views.py`.

Plik `learning_logs/views.py`:

```
--cięcie--  
@login_required  
def topics(request):  
    """Wyświetlenie wszystkich tematów."""  
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')  
    context = {'topics': topics}  
    return render(request, 'learning_logs/topics.html', context)  
--cięcie--
```

Kiedy użytkownik jest zalogowany, obiekt żądania ma ustawiony atrybut `request.user` przechowujący informacje o użytkowniku. Fragment kodu `Topic.objects.filter(owner=request.user)` nakazuje Django pobieranie z bazy danych jedynie tych obiektów `Topic`, których atrybut `owner` odpowiada wartości bieżącego użytkownika. Ponieważ nie zmieniamy sposobu wyświetlania tematów, w ogóle nie ma potrzeby wprowadzania zmian w szablonie dla strony tematów.

Jeżeli chcesz zobaczyć działanie aplikacji po powyższej zmianie, zaloguj się jako użytkownik powiązany z istniejącymi tematami, a następnie przejdź na stronę tematów. Powinieneś zobaczyć wszystkie tematy. Wyloguj się i zaloguj ponownie jako inny użytkownik — strona tematów powinna być pusta.

## Ochrona tematów użytkownika

Tak naprawdę jeszcze nie ograniczyliśmy dostępu do strony tematów, więc zarejestrowany użytkownik może wypróbować różne adresy URL, na przykład `http://localhost:8000/topics/1/` i tym samym pobierać dopasowane strony tematów.

Wypróbuj to sam. Kiedy będziesz zalogowany jako użytkownik, który jest właścicielem wszystkich tematów, skopiuj adres URL lub zanotuj identyfikator dowolnego tematu. Wyloguj się i zaloguj ponownie jako inny użytkownik. Teraz wprowadź skopiowany adres URL tematu. Powinieneś mieć możliwość odczytania wpisów nawet pomimo tego, że jesteś zalogowany jako inny użytkownik.

Usuniemy to niedopatrzenie, przeprowadzając operację sprawdzenia jeszcze przed pobraniem żądanych wpisów w funkcji widoku `topic()`.

Plik `learning_logs/views.py`:

```
from django.shortcuts import render, redirect  
from django.contrib.auth.decorators import login_required  
from django.http import Http404 ①  
  
--cięcie--  
@login_required  
def topic(request, topic_id):  
    """Wyświetla pojedynczy temat i wszystkie powiązane z nim wpisy."""  
    topic = Topic.objects.get(id=topic_id)  
    # Upewniamy się, że temat należy do bieżącego użytkownika.
```

```
if topic.owner != request.user: ❷
    raise Http404

entries = topic.entry_set.order_by('-date_added')
context = {'topic': topic, 'entries': entries}
return render(request, 'learning_logs/topic.html', context)
--cięcie--
```

---

Odpowiedź 404 to standardowa odpowiedź błędu zwracana w sytuacji, gdy żądany zasób nie istnieje w serwerze. W omawianym przykładzie importujemy wyjątek `Http404` (patrz wiersz ❶), który zostanie zgłoszony, jeśli użytkownik zażąda wglądu do tematu należącego do innego użytkownika. Po otrzymaniu żądania tematu, ale jeszcze przed wygenerowaniem strony, upewniamy się, że właściciel tematu i aktualnie zalogowany użytkownik to ta sama osoba. Jeżeli bieżący użytkownik nie będzie właścicielem żądanego tematu, zostanie zgłoszony wyjątek `Http404` (patrz wiersz ❷) i Django zwróci stronę wraz z kodem błędu 404.

Jeżeli teraz spróbujesz wyświetlić temat należący do innego użytkownika, otrzymasz wygenerowany przez Django komunikat o *nieznalezieniu strony*. W rozdziale 20. skonfigurujemy projekt w taki sposób, aby użytkownik otrzymywał odpowiednią stronę błędu.

## Ochrona strony `edit_entry`

Strona `edit_entry` ma adresy URL w postaci `http://localhost:8000/edit_entry/id_wpisu/`, gdzie `id_wpisu` to liczba. Zapewnijmy teraz ochronę tej stronie, aby nieupoważniony użytkownik nie mógł użyć adresu URL w celu uzyskania dostępu do wpisów dokonanych przez innego użytkownika.

*Plik `learning_logs/views.py`:*

---

```
--cięcie--
@login_required
def edit_entry(request, entry_id):
    """Edycja istniejącego wpisu."""
    entry = Entry.objects.get(id=entry_id)
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
--cięcie--
```

---

Pobieramy wpis oraz temat powiązany z danym wpisem. Następnie sprawdzamy, czy właściciel tematu jest aktualnie zalogowanym użytkownikiem. Jeżeli nie, zostaje zgłoszony wyjątek `Http404`.

## Powiązanie nowego tematu z bieżącym użytkownikiem

Obecnie strona przeznaczona do dodawania nowych tematów nie działa zgodnie z oczekiwaniami, ponieważ nie powoduje powiązania nowego tematu z określonym użytkownikiem. Jeżeli spróbujesz dodać nowy temat, otrzymasz komunikat o błędzie spójności (`IntegrityError`), informujący, że `learning_logs_topic.user_id` nie może mieć wartości `NULL`. W ten sposób Django sygnalizuje brak możliwości utworzenia nowego tematu bez podania wartości dla kolumny `owner` tematu.

Rozwiązanie przedstawionego problemu jest całkiem proste, ponieważ dostęp do bieżącego użytkownika mamy za pomocą obiektu `request`. Zmodyfikuj w pokazany poniżej sposób kod funkcji `new_topic()`, a każdy nowo tworzony temat zostanie powiązany z bieżącym użytkownikiem.

Plik `learning_logs/views.py`:

```
--cięcie--  
@login_required  
def new_topic(request):  
    --cięcie--  
    else:  
        # Przekazano dane za pomocą żądania POST, należy je przetworzyć.  
        form = TopicForm(data=request.POST)  
        if form.is_valid():  
            new_topic = form.save(commit=False) ❶  
            new_topic.owner = request.user ❷  
            new_topic.save() ❸  
            return redirect('learning_logs:topics')  
  
    # Wyświetlenie pustego formularza.  
    context = {'form': form}  
    return render(request, 'learning_logs/new_topic.html', context)  
--cięcie--
```

W trakcie pierwszego wywołania `form.save()` przekazujemy argument `commit` `=False`, ponieważ konieczne jest zmodyfikowanie nowego tematu przed jego zapisaniem w bazie danych (patrz wiersz ❶). Następnie atrybutowi `owner` nowego tematu przypisujemy bieżącego użytkownika (patrz wiersz ❷). Na końcu ponownie wywołujemy metodę `save()` w zdefiniowanym przed chwilą egzemplarzu tematu (patrz wiersz ❸). Teraz temat ma wszystkie wymagane dane i zostanie zapisany w bazie danych.

Możesz dodawać dowolną liczbę nowych tematów dla różnych użytkowników. Każdy użytkownik zachowa dostęp jedynie do własnych danych niezależnie od rodzaju przeprowadzanych na nich operacji: przeglądanie wpisów, tworzenie nowych lub modyfikowanie tych już istniejących.

## ZRÓB TO SAM

**19.3. Refaktoryzacja.** W pliku `views.py` mamy dwa miejsca, w których sprawdzamy, czy użytkownik będący właścicielem tematu to aktualnie zalogowany użytkownik. Kod odpowiedzialny za tego rodzaju sprawdzenie umieść w funkcji o nazwie `check_topic_owner()` i wywołuj ją, gdy zajdzie potrzeba.

**19.4. Ochrona strony new\_entry.** Użytkownik może dodać nowy wpis do tematu należącego do innego użytkownika. W tym celu wystarczy podać adres URL wraz z identyfikatorem tematu, którego właścicielem jest inny użytkownik. Zastosuj ochronę przed tego rodzaju atakami i sprawdź przed zapisaniem nowego wpisu, czy bieżący użytkownik jest właścicielem tematu, do którego ma zostać dodany ten wpis.

**19.5. Ochrona bloga.** W budowanym projekcie bloga upewnij się, że każdy post będzie powiązany z określonym użytkownikiem. Zagwarantuj, że wszystkie posty są dostępne publicznie, ale tylko zarejestrowani użytkownicy mogą dodawać nowe posty i edytować te istniejące. Zanim zostanie przetworzony formularz, w widoku pozwalającym użytkownikowi na edycję postu sprawdź, czy użytkownik edytuje własny post.

## Podsumowanie

W tym rozdziale dowiedziałeś się, jak używać formularzy, aby pozwolić użytkownikom na dodawanie nowych tematów i wpisów oraz edytowanie istniejących wpisów. Zobaczyłeś, jak zaimplementować system kont użytkowników. Zarejestrowani użytkownicy mają możliwość zalogować się i wylogować, a niezarejestrowani mogą stworzyć sobie konto (dzięki temu, że użyłeś domyślnego formularza Django o nazwie `UserCreationForm`).

Po zbudowaniu prostego systemu rejestracji i uwierzytelniania użytkowników dostęp do określonych stron przyznaliśmy jedynie zalogowanym użytkownikom. W tym celu wykorzystaliśmy dekoratora `@login_required`. Za pomocą klucza zewnętrznego powiązaliśmy dane z określonymi użytkownikami. Zobaczyłeś, jak przeprowadzić migrację bazy danych, gdy tego rodzaju operacja wymaga podania pewnych danych domyślnych.

Na końcu dowiedziałeś się, jak zmodyfikować funkcje widoku, aby zagwarantować, że użytkownik będzie miał dostęp jedynie do własnych danych. Do pobrania odpowiednich danych wykorzystaliśmy metodę `filter()`. Nauczyłeś się też porównywać właściciela żądanych danych z aktualnie zalogowanym użytkownikiem.

Nie zawsze od razu będzie oczywiste, jakie dane można udostępniać publicznie, a jakie powinny być chronione. Jednak umiejętność właściwej oceny przyjdzie z czasem i praktyką. Podjęte w tym rozdziale decyzje dotyczące ochrony danych użytkowników pokazują, dlaczego praca z innymi jest dobrym rozwiązaniem podczas budowania projektu. Kiedy ktoś inny spojrzeje na Twój projekt, wtedy

prawdopodobnie będzie łatwiej wychwycić ewentualne miejsca, w którym projekt zawiera luki w zabezpieczeniach i tym samym jest podatny na ataki.

W ten sposób przygotowaliśmy w pełni funkcjonalny projekt działający na komputerze lokalnym. W ostatnim rozdziale książki nadamy styl aplikacji *Learning Log*, aby stała się atrakcyjniejsza wizualnie. Ponadto wdrożymy projekt na serwerze i każdy, kto ma dostęp do internetu, będzie mógł zarejestrować się i utworzyć konto w aplikacji.

# 20

## Nadanie stylu i wdrożenie aplikacji



WPRAWDZIE APLIKACJA *LEARNING LOG* JEST JUŻ W PEŁNI FUNKCJONALNA, ALE NIE ZAWIERA ŻADNYCH STYŁÓW I DZIAŁA JEDYNIE NA KOMPUTERZE LOKALNYM. W TYM ROZDZIALE NADAMY APLIKACJI STYL W PROSTY, choć profesjonalny sposób, a następnie wdrożymy ją na serwerze WWW, aby każda osoba na świecie mogła założyć konto w *Learning Log*.

Do nadania stylu aplikacji wykorzystamy bibliotekę Bootstrap. Ta biblioteka to kolekcja narzędzi przeznaczonych do nadawania stylu aplikacjom internetowym, aby prezentowały się profesjonalnie na wszystkich nowoczesnych urządzeniach, począwszy od tych podłączonych do ogromnych płaskich monitorów aż po smartfony. W tym celu będzie nam potrzebna aplikacja django-bootstrap5. Podczas jej używania zdobędziesz doświadczenie związane z wykorzystaniem aplikacji utworzonych przez innych programistów Django.

Przygotowany w książce projekt *Learning Log* wdrożymy za pomocą *Platform.sh*, czyli serwisu pozwalającego na opublikowanie projektu na jednym z jego serwerów i udostępnienie go każdemu, kto tylko ma połączenie z internetem. Rozpoczniemy także używanie systemu kontroli wersji o nazwie Git, aby monitorować zmiany wprowadzane w projekcie.

Kiedy zakończysz pracę nad *Learning Log*, będziesz umiał tworzyć proste aplikacje internetowe, nadawać im elegancki wygląd oraz wdrażać je na serwerze WWW. Ponadto kiedy rozwiniesz swoje umiejętności w zakresie programowania, będziesz mógł korzystać ze znacznie bardziej zaawansowanych zasobów.

# Nadanie stylu aplikacji Learning Log

Wcześniej celowo pominęliśmy kwestie związane ze stylem, aby najpierw skoncentrować się na funkcjonalności aplikacji *Learning Log*. Jest to dobre podejście podczas programowania, ponieważ aplikacja będzie użyteczna tylko wtedy, gdy będzie działać zgodnie z założeniami. Oczywiście kiedy już przygotujemy niezbędne funkcjonalności aplikacji, musimy dopracować jej wygląd tak, aby skłonił użytkowników do korzystania z niej.

W tym podrozdziale przedstawię aplikację `django-bootstrap5` i pokażę, jak można ją zintegrować z projektem. Następnie wykorzystam ją do nadania stylu poszczególnym stronom projektu, aby wszystkie miał spójny wygląd i sposób działania.

## Aplikacja `django-bootstrap5`

Do zintegrowania biblioteki Bootstrap z budowanym projektem wykorzystamy aplikację `django-bootstrap5`. Ta aplikacja pobierze wymagane pliki Bootstrapa, umieści je w odpowiednich katalogach projektu, a następnie udostępnii w szablonach projektu dyrektywy umożliwiające nadawanie stylów.

Aby zainstalować aplikację `django-bootstrap5`, wydaj poniższe polecenie w aktywnym środowisku wirtualnym:

---

```
(11_env)learning_log$ pip install django-bootstrap5
--cięcie--
Successfully installed beautifulsoup4-4.11.1 django-bootstrap5-21.3
soupsieve-2.3.2.post1
```

---

Następnym krokiem jest dodanie poniższego kodu, aby aplikację `django-bootstrap5` umieścić na liście `INSTALLED_APPS` w pliku `settings.py`.

*Plik settings.py:*

---

```
--cięcie--
INSTALLED_APPS = (
    --cięcie--
    # Moje aplikacje.
    'learning_logs',
    'accounts',

    # Aplikacje innych firm.
    'django_bootstrap5',

    # Domyślne aplikacje Django.
    'django.contrib.admin',
)
--cięcie--
```

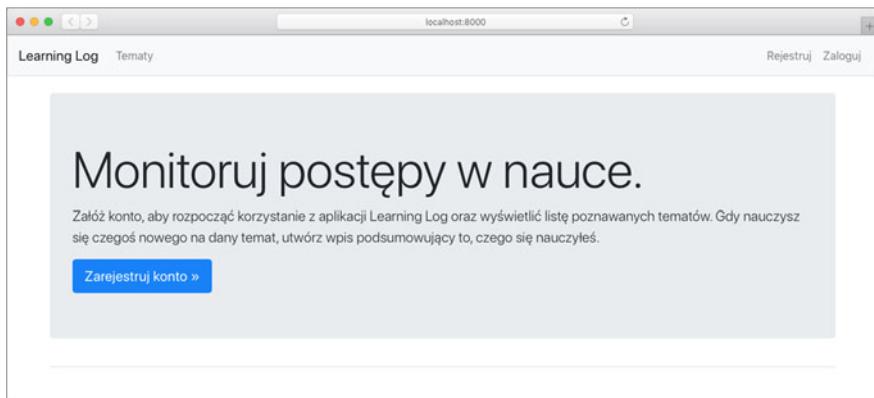
---

Utwórz nową sekcję zatytułowaną na przykład *Aplikacje innych firm* i przeznaczoną na aplikacje utworzone przez innych programistów, a następnie umieść w niej 'django\_bootstrap5'. Upewnij się, że ta sekcja została umieszczona po # Moje aplikacje, ale jeszcze przed domyślnymi aplikacjami Django.

## Użycie Bootstrapa do nadania stylu aplikacji Learning Log

Bootstrap to w zasadzie ogromna kolekcja narzędzi służących do nadawania stylów. Oferuje pewną liczbę szablonów, które można zastosować w projekcie i tym samym osiągnąć konkretny styl. Znacznie łatwiej Ci będzie używać tych szablonów zamiast poszczególnych narzędzi do nadawania stylu. Jeśli chcesz przejrzeć szablony oferowane przez Bootstrapa, przejdź do sekcji *Examples* w witrynie <http://getbootstrap.com/>. W naszym projekcie wykorzystamy szablon *Navbar static*, który dostarczy nam prosty pasek nawigacji na górze strony oraz kontener przeznaczony na treść strony.

Na rysunku 20.1 pokazalem wygląd strony głównej po zastosowaniu szablonu Bootstrapa w pliku *base.html* oraz po przeprowadzeniu drobnej modyfikacji pliku *index.html*.



Rysunek 20.1. Strona główna Learning Log używająca szablonu Bootstrapa

### Modyfikacja pliku base.html

Najpierw musimy zmodyfikować *base.html*, aby móc wykorzystać szablon Bootstrapa. Nową wersję pliku *base.html* przedstawię we fragmentach. To jest ogromny plik i znajdziesz go w materiałach do książki, które zamieściłem pod adresem [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Jeżeli skopiujesz ten plik z materiałów, zapoznaj się z informacjami w tym punkcie, aby zrozumieć zmiany wprowadzone w pliku *base.html*.

## Zdefiniowanie nagłówków HTML

Pierwsza zmiana w pliku *base.html* polega na zdefiniowaniu nagłówków HTML w pliku. Konieczne jest również zdefiniowanie pewnych wymagań do użycia Bootstrapa w szablonach oraz nadanie tytułu stronie. Usuń zawartość pliku *base.html* i zastąp ją poniższym fragmentem kodu.

*Plik base.html:*

---

```
<!doctype html> ①
<html lang="pl"> ②
<head> ③
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Learning Log</title> ④

  {% load django_bootstrap5 %} ⑤
  {% bootstrap_css %}
  {% bootstrap_javascript %}

</head>
```

---

Pierwszym krokiem jest zadeklarowanie pliku jako dokumentu HTML (patrz wiersz ①) zawierającego tekst w języku polskim (patrz wiersz ②). Plik HTML jest podzielony na dwie główne części: *nagłówek* i *treść*, nagłówek rozpoczyna się w wierszu ③. Nagłówek w dokumencie HTML nie zawiera żadnej treści, jedynie informuje przeglądarkę internetową, jak prawidłowo wyświetlić tę stronę. W wierszu ④ mamy element *<title>* definiujący tytuł, który jest wyświetlany na pasku tytułu przeglądarki, kiedy otworzymy dowolną stronę aplikacji *Learning Log*.

Przed zakończeniem sekcji nagłówka wczytujemy kolekcję niestandardowych szablonów znaczników *django-bootstrap5* (patrz wiersz ⑤). Szablon znaczników *{% bootstrap\_css %}* to niestandardowy znacznik z *django-bootstrap5*, który nakazuje Django dodać wszystkie pliki stylów Bootstrapa. Następny znacznik włącza całe interaktywne zachowanie, które może być używane na stronie, na przykład zwijające się paski nawigacyjne. W ostatnim wierszu mamy znacznik zamkający nagłówek — *</head>*.

Wszystkie opcje dotyczące stylów Bootstrapa są teraz dostępne w każdym szablonie dziedziczącym po *base.html*. Jeżeli chcesz użyć niestandardowych szablonów znaczników z *django-bootstrap5*, szablon strony będzie musiał zawierać znacznik *{% load django \_bootstrap5 %}*.

## Zdefiniowanie paska nawigacji

Kod przeznaczony do zdefiniowania paska nawigacji na górze strony jest dość obszerny, ponieważ musi zapewnić obsługę wąskich ekranów w smartfonach i szerokich w tradycyjnych komputerach. Omówienie kodu tworzącego ten pasek znajdziesz w kolejnych sekcjach.

Oto część pierwsza kodu paska nawigacyjnego.

*Plik base.html:*

---

```
--cięcie--  
</head>  
<body>  
  
<nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border"> ❶  
  <div class="container-fluid">  
    <a class="navbar-brand" href="{% url 'learning_logs:index'%}"> ❷  
      Learning Log</a>  
  
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" ❸  
      data-bs-target="#navbarCollapse" aria-controls="navbarCollapse"  
      aria-expanded="false" aria-label="Toggle navigation">  
      <span class="navbar-toggler-icon"></span>  
    </button>  
  
    <div class="collapse navbar-collapse" id="navbarCollapse"> ❹  
      <ul class="navbar-nav me-auto mb-2 mb-md-0"> ❺  
        <li class="nav-item"> ❻  
          <a class="nav-link" href="{% url 'learning_logs:topics' %}"> ❼  
            Tematy</a></li>  
        </ul> <!-- Koniec sekcji łączy po lewej stronie paska nawigacyjnego. -->  
      </div> <!-- Koniec rozwijanych elementów paska nawigacyjnego. -->  
    </div> <!-- Koniec kontenera paska nawigacyjnego. -->  
  </nav> <!-- Koniec paska nawigacyjnego. -->  
  
  {% block content %}{% endblock content %} ❽  
  
</body>  
</html>
```

---

Pierwszym elementem jest otwierający znacznik `<body>`. *Treść* pliku HTML to informacje wyświetlane użytkownikowi na stronie. Następnie mamy element `<nav>` wskazujący na początek sekcji zawierającej łącza nawigacyjne, które mają znaleźć się na stronie (patrz wiersz ❶). Wszystko to, co zostanie umieszczone w tym elemencie, otrzyma styl zgodnie z regulami stylów Bootstrapa zdefiniowanymi przez selektory `navbar`, `navbar-expand-md` i pozostałe, które zostały wymienione w kodzie. Wspomniany *selektor* określa, do których elementów na stronie mają zastosowanie określone reguły stylów. Selektory `navbar-light` i `bg-light` nadają paskowi nawigacyjnemu styl z jasnym tłem. Z kolei `mb` w `mb-4` to skrót od `margin-bottom`, czyli selektora zapewniającego niewielką ilość miejsca między paskiem nawigacyjnym i pozostałą częścią strony. Selektor `border` powoduje wyświetlenie cienkiego obramowania wokół jasnego tła, aby w ten sposób odróżnić je nieco od pozostałej części strony.

Znacznik `<div>` w następnym wierszu powoduje otworzenie kontenera o zmiennej wielkości, przeznaczonego do przechowywania całego paska nawigacyjnego.

Słowo *div* jest skrótem od *division* (z ang. *podział*) — stronę internetową tworzysz przez podzielenie jej na sekcje oraz zdefiniowanie reguł i sposobu działania poszczególnych sekcji. Reguły dotyczące stylu i sposobu działania zdefiniowane w znaczniku otwierającym `<div>` mają zastosowanie dla wszystkich elementów aż do znacznika zamkajającego `</div>`.

W wierszu ② definiujemy nazwę projektu, Learning Log, wyświetlaną jako pierwszy element paska nawigacyjnego i będącą łączem prowadzącym do strony głównej, podobnie jak w pozbawionej stylów wersji projektu utworzonej w dwóch poprzednich rozdziałach. Selektor `navbar-brand` nadaje temu łączu styl odróżniający go od pozostałych łączy i jest często stosowany do wyświetlania nazwy aplikacji w witrynie internetowej.

W wierszu ③ zostaje zdefiniowany przycisk, który będzie wyświetlany, gdy okno przeglądarki będzie miało zbyt małą szerokość, aby wyświetlić poziomo cały pasek nawigacji. Kiedy przycisk ten zostanie kliknięty przez użytkownika, elementy nawigacyjne pojawią się na rozwijanej liście. Reguła `collapse` oznacza zwinięcie się paska nawigacyjnego, gdy użytkownik zmniejszy okno przeglądarki internetowej lub gdy wyświetli witrynę internetową na urządzeniu mobilnym wyposażonym w mały ekran.

W wierszu ④ otwieramy nową sekcję paska nawigacyjnego. To jest początek tej części paska nawigacyjnego, która będzie zwinięta w przypadku wyświetlenia strony na wąskim ekranie lub w wąskim oknie.

Bootstrap definiuje elementy nawigacyjne jako elementy listy nieuporządkowanej (patrz wiersz ⑤) z regułami stylów, dzięki którym nie przypominają listy. Każde łącze lub element, który będzie potrzebny na pasku, można dodać jako element tej listy (patrz wiersz ⑥). W omawianym przykładzie jedynym elementem jest łącze prowadzące na stronę *Tematy* (patrz wiersz ⑦). Zwróć uwagę na znacznik zamkający `</li>` na końcu łącza — dla każdego znacznika otwierającego powinien istnieć odpowiadający mu znacznik zamkający.

Pozostałe wiersze w kodzie zawierają znaczniki zamkające dla wszystkich otworzonych wcześniej znaczników. Komentarz w kodzie HTML ma następującą postać:

---

```
<!-- To jest komentarz w kodzie HTML. -->
```

---

Znaczniki zamkające zwykle nie mają komentarzy. Jeżeli dopiero zaczynasz tworzenie kodu HTML, naprawdę pomocne będzie dodawanie takich komentarzy do znaczników zamkających. Brak nawet tylko jednego znacznika bądź dołączenie nadmiarowego może zniszczyć układ całej strony. W wierszu ⑧ znajduje się blok `content` oraz znaczniki zamkające `</body>` i `</html>`.

Na tym nie kończy się budowa paska nawigacyjnego, ale teraz mamy pełny dokument HTML. Jeżeli działa polecenie `runserver`, zatrzymaj serwer i uruchom go ponownie. Po przejściu na stronę główną projektu zobaczysz pasek nawigacyjny z wybranymi elementami, jak pokazałem na rysunku 20.1. Dodajmy pozostałe elementy do paska nawigacyjnego.

## Zdefiniowanie łączy dotyczących konta użytkownika

Trzeba dodać łącza związane z kontem użytkownika. Rozpoczniemy od dodania wszystkich łączy z wyjątkiem formularza wylogowania.

W pliku *base.html* wprowadź następujące zmiany:

Plik *base.html*:

```
--cięcie--  
    </ul> <!-- Koniec sekcji łączy po lewej stronie paska nawigacyjnego. -->  
  
    <!-- Łącza związane z kontem użytkownika. -->  
    <ul class="navbar-nav ms-auto mb-2 mb-md-0"> ❶  
  
        {% if user.is_authenticated %} ❷  
            <li class="nav-item">  
                <span class="navbar-text me-2"> Witaj, {{ user.username }}. ❸  
            </span></li>  
        {% else %} ❹  
            <li class="nav-item">  
                <a class="nav-link" href="{% url 'accounts:register' %}>  
                    Rejestruj</a></li>  
            <li class="nav-item">  
                <a class="nav-link" href="{% url 'accounts:login' %}>  
                    Zaloguj</a></li>  
        {% endif %}  
  
    </ul> <!-- Koniec łączy związanych z kontem użytkownika. -->  
  
    </div> <!-- Koniec rozwijanych elementów paska nawigacyjnego. -->  
--cięcie--
```

W wierszu ❶ mamy nowy zestaw łączy zdefiniowany po znaczniku otwierającym `<ul>`. Na stronie może znajdować się dowolna liczba grup łączy. Selektor `ms-auto` jest skrótem od `margin-start-automatic` – ten selektor analizuje inne elementy na pasku nawigacyjnym i ustala wielkość lewego (początkowego) marginesu, który powoduje przesunięcie grupy łączy do prawej strony okna przeglądarki WWW.

Blok `if` w wierszu ❷ jest dokładnie tym samym, który wcześniej został użyty do wyświetlenia odpowiedniego komunikatu użytkownikowi w zależności od tego, czy jest on zalogowany, czy nie. Obecnie ten blok jest nieco dłuższy ze względu na reguły stylu umieszczone wewnętrz znaczników warunkowych. W wierszu ❸ znajduje się element `<span>` zawierający komunikat powitania dla uwierzytelnionych użytkowników. Ten *element span* nadaje style fragmentom tekstu, czyli elementom strony, będącym częścią dłuższego wiersza. Podczas gdy znacznik `<div>` powoduje zdefiniowanie oddzielnego fragmentu na stronie, to element `<span>` jest kontynuacją wewnętrz większej sekcji. Na początku to może wydawać się zagmatwane, ponieważ wiele stron ma strukturę bardzo zagnieżdzonych elementów `<div>`. W omawianym przykładzie element `<span>` został użyty do nadania stylu tekstowi informacyjnemu na pasku nawigacyjnym, tutaj do wyświetlania imienia zalogowanego użytkownika.

W bloku `else` wykonywanym dla użytkownika nieuwierzytelnionego zostają umieszczone łącza pozwalające przejść na stronę rejestracji nowego konta bądź logowania ④. Będą one wyglądały podobnie jak łącza na stronie tematów.

Jeżeli chcesz dodać więcej łączy do paska nawigacyjnego, możesz zdefiniować kolejny element `<li>` w dowolnej grupie `<ul>` istniejącej na pasku nawigacyjnym. W tym celu należy skorzystać z identycznych dyrektyw składni jak te, które zostały tutaj omówione.

W następnym podpunkcie dodamy formularz wylogowania do paska nawigacyjnego.

## Dodawanie formularza wylogowania do paska nawigacyjnego

Gdy formularz wylogowania został utworzony, umieściliśmy go na dole strony zdefiniowanej w pliku `base.html`. Teraz umieścimy go w znacznie lepszym miejscu, czyli na pasku nawigacyjnym.

Plik `base.html`:

```
--cięcie--  
</ul> <!-- Koniec łączy związanych z kontem użytkownika. -->  
  
{% if user.is_authenticated %}  
    <form action="{% url 'accounts:logout' %}" method='post'>  
        {% csrf_token %}  
        <button name='submit' class='btn btn-outline-secondary btn-sm'> ❶  
            Wyloguj</button>  
    </form>  
    {% endif %}  
  
</div> <!-- Koniec rozwijanych elementów paska nawigacyjnego. -->  
--cięcie--
```

Formularz wylogowania powinien być umieszczony po zbiorze łączy związanych z kontem użytkownika, ale w sekcji rozwijanych elementów paska nawigacyjnego. Jedyna zmiana w formularzu polega na dodaniu w elemencie `<button>` pewnej liczby klas Bootstrapa, które powodują nadanie stylu przyciskowi wylogowania (patrz wiersz ❶).

Odśwież stronę główną, a zobaczysz, że możesz się zalogować na utworzone konta i z nich wylogować.

To nie koniec dodawania kodu do szablonu `base.html`. Konieczne jest zdefiniowanie dwóch bloków, których poszczególne strony będą mogły używać do umieszczania treści tych stron.

## Zdefiniowanie części głównej strony

Pozostała część pliku `base.html` zawiera część główną strony:

```
--cięcie--  
</nav> <!-- Koniec paska nawigacyjnego. -->  
  
<main class="container"> ❶  
  <div class="pb-2 mb-2 border-bottom"> ❷  
    {% block page_header %}{% endblock page_header %}  
  </div>  
  <div> ❸  
    {% block content %}{% endblock content %}  
  </div>  
</main>  
  
</body>  
</html>
```

---

W wierszu ❶ mamy otwierający znacznik `<main>`. Znacznik `<main>` jest stosowany dla najważniejszej sekcji treści strony. Tutaj następuje przypisanie selektora Bootstrapa o nazwie `container`, który zapewnia prosty sposób grupowania elementów na stronie. W tym kontenerze zostaną umieszczone dwa elementy `<div>`.

Pierwszy z nich zawiera blok o nazwie `page_header` (patrz wiersz ❷). Blok ten zostanie wykorzystany do zdefiniowania tytułu na większości stron. Aby ta sekcja odróżniała się od pozostałej części strony, zdefiniujemy dopełnienie pod nagłówkiem. *Dopełnienie* odwołuje się do wolnego miejsca między treścią elementu i jego obramowaniem. Selektor `pb-2` to dyrektywa Bootstrapa dostarczająca średniej wielkości dopełnienie poniżej elementu, dla którego został zdefiniowany styl. *Margines* to ilość wolnego miejsca między obramowaniem elementu i pozostałymi elementami na stronie. Selektor `mb-2` dostarcza średniej wielkości margines poniżej elementu, dla którego został zdefiniowany styl. Chcemy użyć obramowania tylko na dole strony, stąd zastosowanie selektora `border-bottom`, który zapewnia cienkie obramowanie w dolnej części bloku `page_header`.

W wierszu ❸ został zdefiniowany jeszcze jeden element `<div>` zawierający blok `content`. Nie będziemy stosować żadnego konkretnego stylu dla tego bloku, co pozwoli nadać styl poszczególnym stronom. Na końcu pliku `base.html` znajdują się znaczniki zamkujące elementów `<main>`, `<body>` i `<html>`.

Kiedy wczytasz stronę główną aplikacji *Learning Log* w przeglądarce internetowej, powinieneś zobaczyć profesjonalnie prezentujący się pasek nawigacyjny dopasowany do jednego z pokazanych na rysunku 20.1. Spróbuj zwęzić okno przeglądarki internetowej do naprawdę wąskiego, a zobaczysz, jak pasek nawigacji zostaje zastąpiony przez przycisk. Kiedy klikniesz ten przycisk, wszystkie łącza powinny pojawić się w postaci rozwijanej listy.

# Użycie elementu Jumbotron do nadania stylu stronie głównej

Przystępujemy do uaktualnienia strony głównej za pomocą elementu Bootstrapa o nazwie jumbotron, czyli ogromnego prostokąta wyróżniającego się na tle strony i zawierającego dowolną treść. Ten element jest najczęściej stosowany na stronie głównej i zawiera krótki ogólny opis projektu oraz zachęca użytkownika do podjęcia działania.

Poniżej przedstawiłem zmodyfikowaną wersję pliku *index.html*.

*Plik index.html:*

---

```
{% extends "learning_logs/base.html" %}

{% block page_header %} ❶
<div class="p-3 mb-4 bg-light border rounded-3"> ❷
    <div class="container-fluid py-4">
        <h1 class="display-3">Monitoruj postępy w nauce</h1> ❸

        <p class="lead">Załóż konto, aby rozpocząć korzystanie z aplikacji ❹
            Learning Log oraz wyświetlić listę poznawanych tematów. Gdy
            nauczysz się czegoś nowego na dany temat, utwórz wpis
            podsumowujący to, czego się nauczyłeś.</p>

        <a class="btn btn-primary btn-lg mt-1" href="{% url 'accounts:register' %}"> ❺
            Zarejestruj konto &gt;
        </a>
    </div>
</div>
{% endblock page_header %}
```

---

W wierszu ❶ informujemy Django, że przystępujemy do zdefiniowania treści bloku *page\_header*. Element jumbotron to po prostu para elementów `<div>` z zastosowanym zestawem reguł stylu (patrz wiersz ❷). Zewnętrzny element `<div>` ma ustawienia dopełnienia i marginesu oraz jasny kolor tła i zaokrąglone narożniki. Z kolei wewnętrzny element `<div>` jest kontenerem o wielkości zmieniającej się wraz z wielkością okna oraz z pewnym dopełnieniem. Selektor `py-4` dodaje dopełnienie na górze i dole elementu. Poeksperymentuj z liczbami w tych selektorach i zobacz, jak zmieni się strona główna.

Wewnątrz elementu jumbotron znajdują się trzy inne elementy. Pierwszy to krótki komunikat, *Monitoruj postępy w nauce*, który wskazuje osobom po raz pierwszy odwiedzającym Learning Log, do czego służy ta aplikacja (patrz wiersz ❸). Klasa `h1` to nagłówek pierwszego stopnia, a selektor `display-3` sprawia, nagłówek ten jest cieńszy i wyższy. W wierszu ❹ mamy nieco dłuższy komunikat dostarczający kolejne informacje o tym, do czego użytkownik może wykorzystać tę aplikację. Te informacje są sformatowane jako akapit o klasie `lead`, która powoduje, że powinien się on odróżniać od zwykłych akapitów.

Zamiast używać łącza tekstuowego tworzymy w wierszu ❺ przycisk zachęcający użytkownika do utworzenia konta w aplikacji. To jest to samo łącze, które znajduje

się w nagłówku, ale przycisk wyróżnia się na stronie i wyraźnie pokazuje użytkownikowi, co należy zrobić, aby rozpocząć pracę z aplikacją. Wykorzystane w tym miejscu selektory nadają styl w postaci ogromnego przycisku wzywającego do podjęcia działania. Kod &raquo; to tzw. *encja HTML*, która wygląda jak dwa połączone ze sobą nawiasy ostre (>>). Dalej następuje zamknięcie bloku page\_header. Skoro w pliku są tylko dwa elementy <div>, nie trzeba opisywać komentarzami ich znaczników zamykających. Nie będziemy umieszczać żadnej dodatkowej treści na tej stronie, więc nie trzeba definiować na niej bloku content.

Strona główna wygląda teraz tak, jak pokazałem wcześniej na rysunku 20.1. Jest zdecydowanie lepsza niż pierwotna wersja projektu pozbawiona stylów.

## Nadanie stylu stronie logowania

Dopracowaliśmy ogólny wygląd strony logowania, ale jeszcze nie sam formularz logowania. Przystępujemy więc do wprowadzenia zmian w formularzu logowania, aby zapewnić mu spójny wygląd z pozostałymi stronami.

Plik login.html:

```
% extends "learning_logs/base.html"
% load django_bootstrap5 % ❶

% block page_header % ❷
<h2>Zaloguj się do konta</h2>
% endblock page_header %

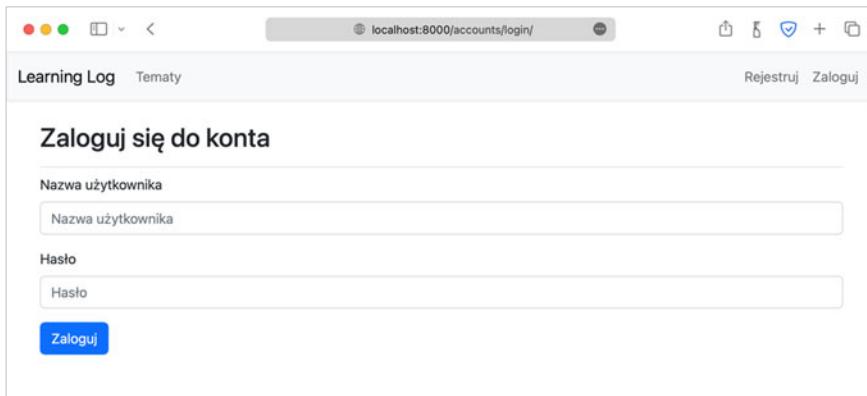
% block content %
<form method="post" action="{% url 'accounts:login' %}">
    {% csrf_token %}
    {% bootstrap_form form %} ❸
    {% bootstrap_button button_type="submit" content="Zaloguj" %} ❹
</form>

% endblock content %
```

W wierszu ❶ wczytujemy do tego szablonu znaczniki szablonu bootstrap5. Dalej w wierszu ❷ definiujemy blok page\_header, który opisuje przeznaczenie tej strony. Zwróć uwagę na usunięcie bloku {% if form.errors %} z szablonu — django-bootstrap5 automatycznie zarządza błędami formularza.

W wierszu ❸ używamy szablonu znacznika {% bootstrap\_form %} do wyświetlenia formularza. Ten znacznik zastępuje znacznik {{ form.as\_div }}, którego używaliśmy w poprzednim rozdziale. Szablon znacznika {% bootstrap\_form %} powoduje wstawienie reguł stylów Bootstrapa do poszczególnych elementów formularza podczas jego generowania. W celu wygenerowania przycisku wysyłającego formularz używamy znacznika {% bootstrap\_button %} z argumentami wskazującymi na przycisk wysłania formularza, któremu nadajemy etykietę Zaloguj (patrz wiersz ❹).

Na rysunku 20.2 pokazałem formularz logowania w postaci wygenerowanej po wprowadzeniu omówionych zmian. Strona stała się bardziej przejrzysta, charakteryzuje się spójnym stylem i jest czytelna (od razu wiadomo, do czego służy). Spróbuj się zalogować, podając nieprawidłowe dane uwierzytelniające (nazwę użytkownika lub hasło), a zobaczysz, że komunikaty błędów mają styl spójny ze stroną i doskonale współgrają z ogólnym wyglądem witryny internetowej.



Rysunek 20.2. Strona logowania wraz ze stylami nadanymi przez Bootstrapa

## Nadanie stylu stronie tematów

Musimy się teraz upewnić, że strony przeznaczone do wyświetlania informacji również mają nadany odpowiedni styl. Rozpoczynamy od strony tematów.

Plik topics.html:

---

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
    <h1>Tematy</h1> ①
{% endblock page_header %}

{% block content %}
    <ul class="list-group border-bottom pb-2 mb-4"> ②
        {% for topic in topics %}
            <li class="list-group-item border-0"> ③
                <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic.text }}</a>
            </li>
        {% empty %}
            <li class="list-group-item border-0">Nie został jeszcze dodany żaden temat.</li> ④
        {% endfor %}
    </ul>

    <a href="{% url 'learning_logs:new_topic' %}">Dodaj nowy temat</a>
{% endblock content %}
```

---

Nie potrzebujemy znacznika `{% load bootstrap5 %}`, ponieważ w tym pliku nie używamy żadnego z własnych znaczników szablonów `bootstrap5`. Wewnątrz bloku `page_header` dodajemy nagłówek *Tematy* (patrz wiersz ❶) i nadajemy mu styl nagłówka zamiast używać zwykłego akapitu.

Treść główna na tej stronie to lista tematów, więc używamy komponentu *listy grupy* Bootstrapa w celu wygenerowania strony. To powoduje zastosowanie prostego zbioru dyrektyw stylu zarówno dla ogólnej listy, jak i jej poszczególnych elementów. W znaczniku otwierającym `<ul>` najpierw pojawia się klasa `list-group` w celu zastosowania domyślnych dyrektyw stylu dla listy (patrz wiersz ❷). Następnie dostosowujemy listę do własnych potrzeb przez umieszczenie obramowania u dołu listy, niewielkiego dopełnienia pod listą (`pb-2`) i marginesu pod dolnym obramowaniem (`mb-4`).

Każdy element na liście wymaga klasy `list-group-item` i modyfikujemy styl domyślny przez usunięcie obramowania wokół poszczególnych elementów (patrz wiersz ❸). Komunikat wyświetlany, gdy lista jest pusta, również otrzymuje styl nadany za pomocą tych samych klas (patrz wiersz ❹).

Gdy teraz odwiedzisz stronę tematów, zobaczysz, że jej styl odpowiada stylowi strony głównej.

## Nadanie stylów wpisom na stronie tematu

Na stronie tematu wykorzystamy karty Bootstrapa, aby wyróżnić każdy wpis. Wspomniana *karta* to po prostu zagnieżdżony zbiór znaczników `<div>` wraz z elastycznymi i predefiniowanymi stylami — doskonale nadaje się do wyświetlania wpisów dla danego tematu.

Plik `topic.html`:

---

```
{% extends 'learning_logs/base.html' %}

{% block page_header %} ❶
    <h1>{{ topic }}</h1>
{% endblock page_header %}

{% block content %}
    <p>
        <a href="{% url 'learning_logs:new_entry' topic.id %}">Dodaj nowy wpis</a>
    </p>

    {% for entry in entries %}
        <div class="card mb-3"> ❷
            <!-- Nagłówek karty razem ze znacznikiem czasu i łączem umożliwiającym
            -->edycję. -->
            <h4 class="card-header"> ❸
                {{ entry.date_added|date:'d M Y H:i' }}
                <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">Edytuj wpis</a></small> ❹
            </h4>
            <!-- Treść karty razem z tekstem wpisu. -->
        </div>
    {% endfor %}
</div>
```

```
<div class="card-body">{{ entry.text|linebreaks }}</div> ⑤  
</div>  
{% empty %}  
<p>Nie ma jeszcze żadnego wpisu dla tego tematu. </p> ⑥  
{% endfor %}  
  
{% endblock content %}
```

---

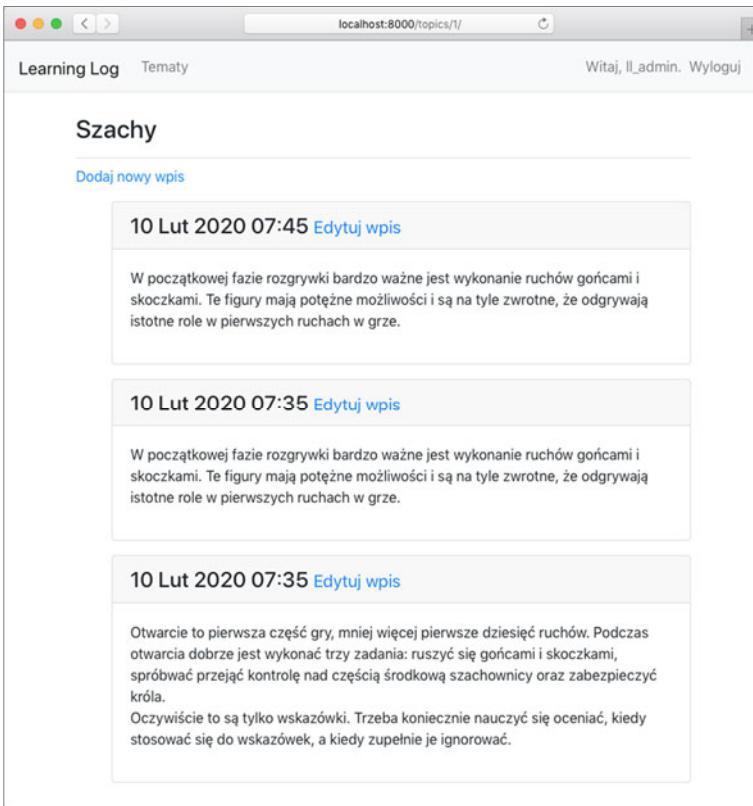
Przede wszystkim temat umieszczamy w bloku `page_header` (patrz wiersz ①). Następnie pozbywamy się struktury nieuporządkowanej listy, jaką poprzednio używaliśmy w tym szablonie. Zamiast nadawać każdemu wpisowi strukturę elementu listy tworzymy element `<div>` karty z selektorem `card` (patrz wiersz ②), która zawiera dwa następne zagnieżdżone elementy `<div>`: pierwszy zawiera datę wpisu oraz łącze umożliwiające jego edycję, natomiast drugi zawiera treść wpisu. Selektor `card` służy do nadawania większości stylów niezbędnych dla tego elementu `<div>`. Kartę dostosowujemy do własnych potrzeb przez dodanie małego marginesu na dole (`mb-3`).

Pierwszy element na karcie to nagłówek, czyli element `<h4>` z selektorem `card-header` (patrz wiersz ③). Ten nagłówek zawiera datę utworzenia wpisu i łącze pozwalające na jego edycję. Dodaliśmy znaczniki `<small>` wokół łącza `edit_entry`, aby było nieco mniejsze niż znacznik czasu (patrz wiersz ④). Drugi element `<div>` ma selektor `card-body` (patrz wiersz ⑤) i powoduje umieszczenie treści wpisu w prostej ramce. Zwróć uwagę, że kod Django odpowiedzialny za umieszczanie informacji na stronie w ogóle nie uległ zmianie. Modyfikujemy tylko elementy wpływające na wygląd strony. Skoro nie używamy już listy nieuporządkowanej, znaczniki elementu listy dla komunikatu informującego o braku elementów do wyświetlenia zastępujemy znacznikami zwykłego akapitu (patrz wiersz ⑥).

Na rysunku 20.3 pokazałem stronę tematu po wprowadzonych zmianach. Funkcjonalność aplikacji *Learning Log* się nie zmieniła, ale sama witryna wygląda teraz znacznie bardziej profesjonalnie i zachęcająco dla użytkowników.

Jeżeli chcesz użyć zupełnie innego szablonu Bootstrapa, zastosuj proces podobny do przedstawionego w tym rozdziale. Skopiuj szablon do pliku `base.html`, a następnie zmodyfikuj elementy zawierające unikatową treść, aby szablon wyświetlał informacje o projekcie. W kolejnym kroku użyj narzędzi nadawania stylów Bootstrapa, aby nadać style treści wyświetlanej na poszczególnych stronach.

**UWAGA** *Projekt Bootstrap ma doskonałą dokumentację. Odwiedź witrynę <https://getbootstrap.com/> i kliknij Docs, aby dowiedzieć się więcej na temat możliwości Bootstrapa.*



Rysunek 20.3. Strona tematu wraz z zastosowanymi stylami Bootstrapa

## ZRÓB TO SAM

**20.1. Inne formularze.** Style Bootstrapa zastosowaliśmy na stronie login. Wprowadź podobne zmiany w pozostałych stronach opartych na formularzach: new\_topic, new\_entry, edit\_entry i register.

**20.2. Nadanie stylu blogowi.** Wykorzystaj bibliotekę Bootstrap do nadania stylów projektowi bloga, który utworzyłeś w ćwiczeniach do poprzedniego rozdziału.

## Wdrożenie aplikacji Learning Log

Skoro mamy gotowy, profesjonalnie prezentujący się projekt, warto wdrożyć go na serwerze WWW, aby każdy, kto ma połączenie z internetem, mógł skorzystać z naszej aplikacji. Wykorzystamy tutaj Platform.sh — platformę sieciową pozwalającą na zarządzanie operacją wdrażania aplikacji internetowej. Przedstawione tutaj kroki pokażą, jak otrzymać działającą w Platform.sh aplikację *Learning Log*.

## Utworzenie konta w Platform.sh

W celu utworzenia konta przejdź do witryny <https://platform.sh/> i kliknij przycisk *Free Trial*. Serwis Platform.sh oferuje konto próbne, którego utworzenie nie wymaga podawania numeru karty kredytowej (przynajmniej w chwili powstawania książki). Konto próbne umożliwia wdrożenie aplikacji wykorzystującej minimalną ilość zasobów, co pozwala przetestować projekty przed wykupieniem konta płatnego.

**UWAGA** *Konkretnie ograniczenia konta próbnego zmieniają się co jakiś czas, ponieważ platformy hostingowe walczą ze spamem i nadużyciami zasobów. Na stronie <https://platform.sh/free-trial/> możesz zapoznać się z bieżącymi ograniczeniami konta próbnego.*

## Instalacja Platform.sh CLI

Aby wdrożyć projekt i zarządzać nim na serwerach Platform.sh, konieczne będzie użycie narzędzi dostępnych w powłoce (ang. *command line interface*, CLI). By zainstalować najnowszą wersję Platform.sh CLI, przejdź na stronę <https://docs.platform.sh/administration/cli.html>, a następnie wykonuj kolejne kroki zgodnie z instrukcją dla używanego przez Ciebie systemu operacyjnego.

W większości systemów to oznacza konieczność wydania następującego polecenia w powłoce:

---

```
$ curl -fsS https://platform.sh/cli/installer | php
```

---

Po zakończeniu wykonywania tego polecenia trzeba otworzyć nowe okno powłoki, aby można było używać zainstalowanych narzędzi.

**UWAGA** *To polecenie prawdopodobnie nie będzie działało w standardowym wierszu poleceń Windowsa. Możesz wykorzystać powłokę Windows Subsystem for Linux (WSL) lub Git Bash. Jeżeli musisz zainstalować PHP, to możesz wykorzystać instalator XAMPP ze strony <https://www.apachefriends.org/>. W przypadku trudności z instalacją Platform.sh CLI zapoznaj się ze znacznie dokładniejszymi informacjami na temat instalacji, które znajdziesz w dodatku E.*

## Instalacja platformshconfig

Konieczne będzie zainstalowanie pakietu dodatkowego, `platformshconfig`. Pomaga on wykryć, czy projekt działa w systemie lokalnym, czy na serwerze Platform.sh. W aktywnym środowisku wirtualnym wydaj następujące polecenie:

---

```
(11_env)learning_log$ pip install platformshconfig
```

---

Pakiet `platformshconfig` wykorzystamy do modyfikacji ustawień projektu podczas jego działania na serwerze WWW.

## Utworzenie pliku requirements.txt

Zdalny serwer musi wiedzieć, jakie pakiety są niezbędne do działania naszego projektu. Dlatego też wykorzystamy menedżera pip do wygenerowania pliku zawierającego listę zależności projektu. Po przejściu do aktywnego środowiska wirtualnego wydaj poniższe polecenie:

---

```
(11_env)learning_log$ pip freeze > requirements.txt
```

---

Polecenie freeze nakazuje menedżerowi pip zapisanie w pliku *requirements.txt* nazw wszystkich pakietów aktualnie zainstalowanych w projekcie. Po otwarciu wymienionego pliku zobaczysz nazwy pakietów i ich wersje zainstalowane w projekcie.

*Plik requirements.txt:*

---

```
asgiref==3.5.2
beautifulsoup4==4.11.1
Django==4.1
django-bootstrap5==21.3
platformshconfig==2.4.0
soupsieve==2.3.2.post1
sqlparse==0.4.2
```

---

Działanie aplikacji *Learning Log* zależy od siedmiu pakietów w konkretnych wersjach, więc do prawidłowego działania projektu na zdalnym serwerze niezbędne jest określone środowisko. (Trzy z tych pakietów zostały zainstalowane ręcznie, natomiast pozostałe cztery zostały zainstalowane automatycznie jako zależności tych pierwszych).

Kiedy będziemy wdrażać aplikację *Learning Log*, Platform.sh zainstaluje pakiety wymienione w pliku *requirements.txt* i tym samym utworzy środowisko zawierające takie same pakiety, jakie były używane w środowisku lokalnym. Dlatego też mamy pewność, że wdrożony projekt będzie zachowywał się dokładnie w taki sam sposób jak w naszym systemie lokalnym. Takie podejście w zakresie zarządzania projektem ma znaczenie krytyczne, gdy rozpoczniesz budowanie i obsługiwanie różnych projektów w swoim systemie.

**UWAGA** Jeżeli w systemie masz zainstalowany pakiet w innej wersji niż przedstawiona w książce, pozostań przy zainstalowanej wersji pakietu.

## Dodatkowe wymagania dotyczące wdrożenia

Serwer WWW będzie wymagał jeszcze dwóch pakietów dodatkowych. Są one używane do obsługi projektu w środowisku produkcyjnym, w którym jednocześnie wielu użytkowników może wykonywać żądania.

W katalogu zawierającym plik *requirements.txt* utwórz nowy plik o nazwie *requirements\_remote.txt*, a następnie umieść w nim dwa następujące pakiety:

---

```
# Wymagania od uruchomienia projektu w serwerze WWW.  
gunicorn  
psycopg2
```

---

Pakiet *gunicorn* odpowiada na żądania wykonywane do zdalnego serwera. Przejmuje zadania serwera programistycznego, którego używamy lokalnie. Z kolei pakiet *psycopg2* jest wymagany, aby umożliwić Django zarządzanie bazą danych Postgres używaną przez Platform.sh. *Postgres* to baza danych open source, która wyjątkowo świetnie radzi sobie z aplikacjami produkcyjnymi.

## Dodawanie plików konfiguracyjnych

Każda platforma hostingowa wymaga pewnej konfiguracji projektu, aby mógł być on poprawnie uruchamiany na jej serwerach. W tym miejscu dodamy trzy pliki konfiguracyjne:

*.platform/app.yaml* — jest to główny plik konfiguracyjny projektu. Wskazuje Platform.sh rodzaj projektu, który próbujemy wdrożyć, a także rodzaje zasobów wymaganych przez ten projekt. Obejmuje polecenia niezbędne do utworzenia projektu na serwerze.

*.platform/routes.yaml* — definiuje trasy dla projektu. Po otrzymaniu żądania przez Platform.sh ten plik konfiguracyjny pomoże w przekierowaniu żądania do właściwego projektu.

*.platform/services.yaml* — definiuje wszelkie usługi dodatkowe wymagane przez projekt.

Wszystkie trzy pliki są w formacie YAML (YAML Ain't Markup Language). YAML to język opracowany w celu tworzenia plików konfiguracyjnych. Założenia ma być łatwy w odczycie zarówno dla człowieka, jak i dla maszyny. Pliki YAML można tworzyć lub modyfikować ręcznie, komputer również potrafi je bez problemów odczytywać i interpretować.

Pliki YAML świetnie sprawdzają się przy konfigurowaniu wdrożenia, ponieważ zapewniają dobrą kontrolę nad przebiegiem procesu.

## Wyświetlanie plików ukrytych

Większość systemów operacyjnych ukrywa pliki i katalogi o nazwach rozpoczynających się kropką, np. *.platform*. Gdy otworzysz menedżer plików, domyślnie nie zobaczysz plików ukrytych. Jednak jako programista musisz z nich korzystać. Oto jak można wyświetlić pliki ukryte w różnych systemach operacyjnych:

- W systemie Windows przejdź do *Eksploratora plików* i otwórz katalog, np. *Pulpit*. Kliknij kartę *Widok*, a następnie upewnij się, że są zaznaczone opcje *Rozszerzenia nazw plików* i *Ukryte elementy*.
- W systemie macOS naciśnij klawisze *Cmd+Shift+.* (kropka) w dowolnym oknie *Findera*, co spowoduje wyświetlenie ukrytych plików i katalogów.
- W systemie Linux, np. Ubuntu, możesz w dowolnym oknie przeglądarki plików nacisnąć klawisze *Ctrl+H*, co spowoduje wyświetlenie ukrytych plików i katalogów. Aby to ustawienie pozostało trwałe, otwórz przeglądarkę plików taką jak *Nautilus* i kliknij kartę opcji (jej ikona przedstawia trzy linie). Następnie zaznacz pole wyboru *Ukryte pliki*.

## Plik konfiguracyjny `.platform.app.yaml`

Plik konfiguracyjny jest najdłuższy, ponieważ kontroluje ogólny proces wdrożenia. Przedstawię go we fragmentach, które możesz ręcznie wpisać w edytorze tekstu. Ewentualnie pełną wersję pliku pobierz z materiałów przygotowanych dla książki, które znajdziesz pod adresem [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/).

Oto pierwsza część pliku `.platform.app.yaml`, który powinien być zapisany w katalogu zawierającym plik `manage.py`:

*Plik `.platform.app.yaml`:*

```
name: "11_project" ❶
type: "python:3.10"

relationships: ❷
    database: "db:postgresql"

# Konfiguracja aplikacji, gdy nie jest udostępniona w internecie.
web: ❸
    upstream:
        socket_family: unix
    commands:
        start: "gunicorn -w 4 -b unix:$SOCKET 11_project.wsgi:application" ❹
locations: ❺
    "/":
        passthru: true
    "/static":
        root: "static"
        expires: 1h
        allow: true

# Wyrażona w megabajtach ilość miejsca na dysku zarezerwowana dla aplikacji.
disk: 512 ❻
```

Podeczas zapisywania tego pliku upewnij się, że na początku nazwy pliku jest umieszczona kropka. Jeżeli ją pominiesz, Platform.sh nie znajdzie tego pliku i projekt nie zostanie wdrożony.

W tym momencie nie musisz w pełni rozumieć znaczenia kodu zamieszczonego w pliku `.platform.app.yaml`. Wskazałem najważniejsze fragmenty konfiguracji. Plik rozpoczyna się od określenia nazwy projektu, którą jest '11\_project' (patrz wiersz ❶) w celu zachowania spójności z nazwą użytą przy rozpoczętym projekcie. Konieczne jest również określenie używanej wersji Pythona (w chwili powstawania książki 3.10). Listę obsługiwanych przez Platform.sh wersji Pythona znajdziesz pod adresem <https://docs.platform.sh/languages/python.html>.

Następna sekcja jest określona jako `relationships` i definiuje inne usługi niezbędne projektowi (patrz wiersz ❷). W omawianym przykładzie wymagana jest jedynie baza danych Postgres. Dalej znajduje się sekcja `web` (patrz wiersz ❸). W sekcji `commands:start` wskazujemy Platform.sh proces, który ma być wykorzystywany do obsługi żądań przychodzących. W naszej aplikacji będzie to `gunicorn` (patrz wiersz ❹). Podane tutaj polecenie jest używane zamiast `python manage.py runserver`, z którego korzystaliśmy do uruchamiania lokalnej wersji projektu.

Sekcja `locations` wskazuje Platform.sh, gdzie mają zostać przekazane żądania przychodzące (patrz wiersz ❺). Większość tych żądań powinna trafić do `gunicorn`. Plik `urls.py` w projekcie będzie dokładnie wskazywał `gunicorn`, jak mają zostać obsłużone te żądania. Żądania dotyczące plików statycznych będą obsługiwane oddzielnie i odświeżane raz na godzinę. W ostatnim wierszu możesz zobaczyć, że żadamy 512 MB miejsca na dysku twardym serwera Platform.sh ❻.

Pozostała część pliku `.platform.app.yaml` przedstawia się następująco:

#### *Plik .platform.app.yaml:*

---

```
--cięcie--
disk: 512

# Określenie lokalnych punktów montowania dla dzienników zdarzeń.
mounts: ❶
  "logs":
    source: local
    source_path: logs

# Zaczepły wykonywane w różnych punktach cyklu życiowego aplikacji.
hooks: ❷
  build: |
    pip install --upgrade pip ❸
    pip install -r requirements.txt
    pip install -r requirements_remote.txt

    mkdir logs
    python manage.py collectstatic ❹
    rm -rf logs

  deploy: | ❺
    python manage.py migrate
```

---

Sekcja `mounts` (patrz wiersz ①) pozwala zdefiniować katalogi, w których można odczytywać i zapisywać dane podczas działania projektu. W omawianym przykładzie został zdefiniowany katalog `logs/` dla wdrożonego projektu.

Sekcja `hooks` (patrz wiersz ②) definiuje akcje podejmowane na różnych etapach procesu wdrożenia. W sekcji `build` będą instalowane pakiety niezbędne do obsługi projektu w środowisku produkcyjnym (patrz wiersz ③). Wykonywane jest również polecenie `collectstatic` (patrz wiersz ④), którego zadaniem jest pobranie wszystkich plików statycznych wymaganych przez projekt. Te pliki zostaną umieszczone w jednym miejscu, aby mogły być efektywnie udostępniane.

Wreszcie mamy sekcję `deploy` (patrz wiersz ⑤), określającą migracje, które należy przeprowadzić podczas każdego wdrożenia projektu. W takim prostym projekcie jak omawiany nie będzie to miało znaczenia, o ile nie wprowadzono zmian.

Dwa pozostałe pliki konfiguracyjne są znacznie krótsze. Omówię je w kolejnych podpunktach.

## Plik konfiguracyjny `routes.yaml`

*Trasa* (agn. *route*) to ścieżka pokonywana przez żądanie w trakcie jego obsługi przez serwer. Po otrzymaniu żądania serwer `Platform.sh` musi wiedzieć, gdzie je przekazać.

W katalogu zawierającym plik `manage.py` utwórz nowy podkatalog, o nazwie `.platform`. Upewnij się, że na początku jego nazwy znajduje się kropka. Następnie w nowym katalogu umieść plik o nazwie `routes.yaml` i następującej zawartości:

*Plik `.platform/routes.yaml`:*

---

```
# Każda trasa opisuje sposób przetwarzania przez serwer Platform.sh przychodzącego adresu URL.
"https://{{default}}/":
    type: upstream
    upstream: "ll_project:http"

"https://www.{{default}}/":
    type: redirect
    to: "https://{{default}}/"
```

---

Kod w tym pliku gwarantuje, że żądania takie jak `https://adres_url_projektu.com` i `www.adres_url_projektu.com` zostaną przekierowane w to samo miejsce.

## Plik konfiguracyjny `services.yaml`

Ostatni plik konfiguracyjny, `services.yaml`, określa usługi wymagane do prawnego działania projektu. Zapisz go w katalogu `.platform`, razem z plikiem `routes.yaml`.

## Plik .platform/services.yaml:

---

```
# Każda wymieniona tutaj usługa będzie wdrożona w oddzielnym kontenerze
# jako część projektu Platform.sh.
db:
  type: postgresql:12
  disk: 1024
```

---

W pliku została zdefiniowana tylko jedna usługa — bazy danych Postgres.

## Modyfikacja pliku settings.py dla Platform.sh

Na końcu pliku *settings.py* konieczne jest dodanie sekcji zawierającej ustawienia dedykowane dla środowiska Heroku.

### Plik settings.py:

---

```
--cięcie--
# Ustawienia Platform.sh.
from platformshconfig import Config ❶

config = Config()
if config.is_valid_platform(): ❷
    ALLOWED_HOSTS.append('.platformsh.site') ❸

    if config.appDir: ❹
        STATIC_ROOT = Path(config.appDir) / 'static'
    if config.projectEntropy: ❺
        SECRET_KEY = config.projectEntropy

    if not config.in_build():
        db_settings = config.credentials('database') ❻
        DATABASES = {
            'default': {
                'ENGINE': 'django.db.backends.postgresql',
                'NAME': db_settings['path'],
                'USER': db_settings['username'],
                'PASSWORD': db_settings['password'],
                'HOST': db_settings['host'],
                'PORT': db_settings['port'],
            },
        }
```

---

Polecenia `import` normalnie znajdują się na początku modułu. Jednak w tym przypadku znacznie lepszym podejściem będzie umieszczanie w jednej sekcji wszystkich ustawień związanych z danym serwerem zdalnym. W wierszu ❶ importujemy `Config` z modułu `platformshconfig`, co pomoże w ustaleniu ustawień zdalnego serwera. Ustawienia zostaną zmodyfikowane tylko wtedy, gdy metoda `config.is_valid_platform()` zwróci wartość `True` ❷, wskazując tym samym ustawienia używane na serwerze Platform.sh.

Modyfikujemy wartość ALLOWED\_HOSTS, aby umożliwiał obsługę projektu przez hosty o nazwie kończącej się na `.platformsh.site` ❸. Wszystkie projekty wdrażane w ramach konta bezpłatnego zostaną wdrożone z użyciem tego hosta. Jeżeli ustawienia są wczytywane w katalogu wdrożonej aplikacji ❹, definiujemy STATIC\_ROOT, aby pliki statyczne były poprawnie udostępniane. Konieczne jest również zdefiniowanie znacznie bezpieczniejszego klucza, SECRET\_KEY, w zdalnym serwerze ❺.

Na końcu konfigurujemy produkcyjną bazę danych ❻. Ta operacja zostanie przeprowadzona jedynie w przypadku zakończenia procesu komplikacji i działania aplikacji. Wszystkie przedstawione tutaj opcje są niezbędne, aby umożliwić Django komunikację z serwerem Postgres skonfigurowanym przez Platform.sh na potrzeby tego projektu.

## Użycie Gita do monitorowania plików projektu

Jeżeli czytałeś rozdział 17., to wiesz, że Git jest systemem kontroli wersji pozwalającym na utworzenie migawki kodu źródłowego projektu za każdym razem, gdy z powodzeniem zakończysz implementację nowej funkcjonalności. W ten sposób możesz bardzo łatwo przywrócić ostatnią prawidłowo działającą wersję projektu, jeśli cokolwiek pójdzie źle podczas pracy nad nową funkcją, na przykład niechcący wprowadzisz błąd. Każda tego rodzaju migawka jest nazywana *zatwierdzeniem* (*commit*).

Dzięki systemowi Git możesz pracować nad implementacją nowych funkcji bez obaw o uszkodzenie projektu. Kiedy przystępujesz do wdrożenia projektu na serwerze produkcyjnym, musisz się upewnić, że wdrażasz prawidłowo działającą wersję projektu. Jeżeli chcesz dowiedzieć się więcej na temat systemu Git i kontroli wersji, zajrzyj do dodatku D.

## Instalacja Git

Git może być już zainstalowany w Twoim systemie operacyjnym. Aby to sprawdzić, otwórz nowe okno powłoki i wydaj polecenie `git --version`:

```
(11_env)learning_log$ git --version
git version 2.30.1 (Apple Git-130)
```

Jeżeli z jakiegokolwiek powodu otrzymasz komunikat błędu, informacje dotyczące procedury instalacji systemu Git znajdziesz w dodatku D.

## Konfiguracja Git

System Git zapisuje informacje o tym, kto wprowadził zmiany w projekcie, nawet jeśli pracuje nad nim tylko jedna osoba. Dlatego też Git musi znać Twoją nazwę użytkownika i adres e-mail. Nazwę użytkownika musisz podać, natomiast w przypadku adresu e-mail masz większą elastyczność:

---

```
(11_env)learning_log$ git config --global user.name "eric"
(11_env)learning_log$ git config --global user.email "eric@example.com"
```

---

Jeżeli zapomnisz o tym kroku, Git poprosi o podanie tych informacji podczas pierwszej operacji przekazywania kodu do repozytorium.

## Ignorowanie plików

Nie ma potrzeby, aby Git monitorował wszystkie pliki w projekcie. Dlatego też wskażemy teraz te, które powinny być zignorowane. W katalogu zawierającym plik *manage.py* utwórz plik o nazwie *.gitignore*. Zwróć uwagę, że nazwa nowego pliku rozpoczyna się od kropki i nie zawiera rozszerzenia. Poniżej przedstawiłem zawartość tego pliku.

*Plik .gitignore:*

---

```
11_env/
__pycache__/
*.sqlite3
```

---

Nakazujemy systemowi Git zignorowanie całego katalogu *ll\_env*, ponieważ możemy go automatycznie odtworzyć w dowolnym momencie. Ponadto nie monitorujemy katalogu *\_\_pycache\_\_* zawierającego pliki *.pyc*, tworzone automatycznie podczas wykonywania plików *.py*. Nie będziemy monitorować zmian w lokalnej bazie danych, ponieważ to jest zły nawyk. Jeżeli korzystasz z bazy danych SQLite na serwerze, to podczas przekazywania projektu do serwera produkcyjnej bazę danych możesz przypadkowo nadpisać jej lokalną wersją testową. Gwiazdka w poleceniu *\*.sqlite3* nakazuje systemowi Git ignorowanie wszystkich plików z rozszerzeniem *.sqlite3*.

**UWAGA** Jeżeli używasz systemu macOS, do pliku *.gitignore* dodaj jeszcze *.DS\_Store*. Jest to plik przechowujący informacje o ustawieniach katalogu w systemie macOS i nie ma nic wspólnego z omawianym projektem.

## Zatwierdzenie projektu

Konieczne jest zainicjalizowanie repozytorium Git dla aplikacji *Learning Log*, umieszczenie wszystkich niezbędnych plików w repozytorium oraz zatwierdzenie początkowego stanu projektu. Poniżej przedstawiłem tego rodzaju procedurę:

---

```
(11_env)learning_log$ git init ❶
Initialized empty Git repository in /Users/eric/.../learning_log/.git/
(11_env)learning_log$ git add .
❷
(11_env)learning_log$ git commit -am "Projekt gotowy do wdrożenia w Platform.sh." ❸
[main (root-commit) c7ffaad] Projekt gotowy do wdrożenia w Platform.sh.
 42 files changed, 879 insertions(+)
  create mode 100644 .gitignore
```

---

```
create mode 100644 .platform.app.yaml
--cięcie--
create mode 100644 requirements_remote.txt
(11_env)learning_log$ git status ④
On branch main
nothing to commit, working directory clean
(11_env)learning_log$
```

---

W wierszu ① wydajemy polecenie `git init` w celu inicjalizacji pustego repozytorium w katalogu zawierającym aplikację *Learning Log*. Polecenie `git add .` (nie zapomnij o kropce na końcu) w wierszu ② powoduje dodanie do repozytorium wszystkich plików, które nie zostały wskazane jako ignorowane. W wierszu ③ wydajemy polecenie `git commit -am "komunikat"`. Opcja `-a` nakazuje Gitowi uwzględnić wszystkie zmodyfikowane pliki, natomiast opcja `-m` nakazuje zarejestrować podany komunikat.

Polecenie `git status` w wierszu ④ wskazuje, że aktualnie znajdujemy się w gałęzi *main*, a katalog roboczy jest określony jako *czysty (clean)*. Taki stan powinieneś mieć za każdym razem, gdy przekazujesz projekt na zdalny serwer.

## Utworzenie projektu w Platform.sh

Na tym etapie projekt *Learning Log* nadal działa w systemie lokalnym, choć został również skonfigurowany do poprawnego działania na zdalnym serwerze. Oferowane przez Platform.sh narzędzie powłoki wykorzystamy do utworzenia nowego projektu na serwerze, a następnie przekażemy projekt na serwer zdalny.

W powłoce przejdź do katalogu *learning\_log*, a następnie wydaj następujące polecenie:

```
(11_env)learning_log$ platform login
Opened URL: http://127.0.0.1:5000
Please use the browser to log in.
--cięcie--
Do you want to create an SSH configuration file automatically? [Y/n] Y ①
```

---

To polecenie spowoduje otworzenie w oknie przeglądarki WWW nowej karty, w której będzie można się zalogować. Po zalogowaniu należy zamknąć tę kartę i powrócić do powłoki. Jeżeli zobaczyś pytanie o to, czy chcesz utworzyć plik konfiguracyjny SSH (patrz wiersz ①), wpisz `Y` i naciśnij klawisz *Enter*, aby później móc nawiązać połączenie ze zdalnym serwerem.

Teraz utworzymy projekt. Ponieważ ta operacja spowoduje wyświetlenie znacznej ilości danych wyjściowych, omówię ją we fragmentach. Rozpocznij od wydania polecenia `platform create`.

```
(11_env)learning_log$ platform create
* Project title (--title)
Default: Untitled Project
> 11_project ①
```

---

```
* Region (--region)
The region where the project will be hosted
--cięcie--
[us-3.platform.sh] Moses Lake, United States (AZURE) [514 gCO2eq/kWh]
> us-3.platform.sh ②
* Plan (--plan)
Default: development
Enter a number to choose:
[0] development
--cięcie--
> 0 ③

* Environments (--environments)
The number of environments
Default: 3
> 3 ④

* Storage (--storage)
The amount of storage per environment, in GiB
Default: 5
> 5 ⑤
```

---

Pierwsze pytanie dotyczy nazwy projektu (patrz wiersz ①) — użyjemy tutaj nazwy `ll_project`. Następnie trzeba wybrać region, w którym ma się znajdować serwer ② — wybierz najbliższy geograficznie serwer, w moim przypadku jest to `us-3.platform.sh`. W pozostałych pytanach można zaakceptować wartości domyślne: serwer z planem o najmniejszych możliwościach programistycznych ③, trzy środowiska dla projektu ④ oraz 5 GB miejsca na dysku przeznaczonego dla całego projektu ⑤.

Mamy jeszcze trzy pytania, na które trzeba udzielić odpowiedzi.

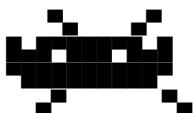
---

```
Default branch (--default-branch)
The default Git branch name for the project (the production environment)
Default: main
> main ①

Git repository detected: /Users/eric/.../learning_log
Set the new project ll_project as the remote for this repository? [Y/n] Y ②

The estimated monthly cost of this project is: $10 USD
Are you sure you want to continue? [Y/n] Y ③

The Platform.sh Bot is activating your project
```



The project is now ready!

---

Repozytorium Git może zawierać wiele gałęzi. W wierszu ❶ Platform.sh pyta, czy domyślną gałęzią dla projektu powinna być `main`. Następne pytanie dotyczy połączenia repozytorium projektu lokalnego ze zdalnym repozytorium (patrz wiersz ❷). Na końcu została podana informacja o miesięcznym koszcie utrzymania tego projektu — w przykładzie jest to około 10 dolarów (patrz wiersz ❸). Możesz zdecydować, czy chcesz go zachować po upływie okresu próbnego. Jeżeli jeszcze nie podałeś danych karty kredytowej, nie musisz się przejmować kosztem projektu. Platform.sh po prostu zawiesi projekt po przekroczeniu ograniczeń dla konta próbnego, nie wymagając przy tym podawania danych karty kredytowej.

## Przekazanie projektu do Platform.sh

Ostatnim krokiem, zanim będzie możliwe użycie aplikacji w internecie, jest przekazanie jej kodu źródłowego na zdalny serwer. W aktywnej sesji powłoki wydaj następujące polecenia:

```
(11_env)learning_log$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] Y ❶
--cięcie--
The authenticity of host 'git.us-3.platform.sh (...)' can't be established.
RSA key fingerprint is SHA256:Tvn...7PM
Are you sure you want to continue connecting (yes/no/[fingerprint])? Y ❷
Pushing HEAD to the existing environment main
--cięcie--
To git.us-3.platform.sh:3pp3mqcexhlyv.git
 * [new branch]      HEAD -> main
```

Po wydaniu polecenia **platform push** pojawi się jeszcze jedna prośba o potwierdzenie operacji przekazania projektu na serwer (patrz wiersz ❶). Możesz również otrzymać komunikat dotyczący uwierzytelnienia w Platform.sh (patrz wiersz ❷), jeśli połączenie z serwerem nawiązujesz po raz pierwszy. Wpisz `Y` jako odpowiedzi na oba te pytania, a zobaczysz wygenerowaną ogromną ilość danych wyjściowych. Prawdopodobnie na początku będą one przytłaczające, ale jeśli coś pójdzie nie tak, dostarczane przez nie informacje okażą się użyteczne podczas rozwiązywania problemów. Pobieżnie przeglądając te dane, można zauważać, gdzie Platform.sh instaluje niezbędne pakiety, jak pobiera pliki statyczne, jak przeprowadza migracje i jak konfiguruje adresy URL dla projektu.

**UWAGA** *Możesz otrzymać błąd z powodu problemu łatwego do zdiagnozowania, takiego jak błąd w jednym z plików konfiguracyjnych. W takim przypadku usuń błąd za pomocą edytora tekstu, zapisz plik, a potem ponownie wydaj polecenia git commit i platform push.*

## Wyświetlenie wdrożonego projektu

Po przeprowadzeniu wdrożenia projekt można uruchomić:

```
(11_env)learning_log$ platform url  
Enter a number to open a URL  
[0] https://main-bvxe6i-wmye2fx7wwqgu.us-3.platformsh.site/  
--cięcie--  
> 0
```

Polecenie platform url wyświetla listę adresów URL powiązanych z wdrożonym projektem. Otrzymasz kilka adresów URL, które są poprawne w danym projekcie. Wybierz jeden, a projekt powinien zostać uruchomiony w przeglądarce WWW. Wprawdzie może się wydawać, że projekt został uruchomiony lokalnie, ale każdy na świecie, kto zna jego adres URL, będzie mógł uzyskać dostęp do tego projektu.

**UWAGA** W przypadku udrożenia projektu z użyciem konta próbnego nie należy się dziwić, że czasami wczytanie strony trwa dłużej niż zwykle. Na większości platform hostingowych wolne dostępne bezpłatnie zasoby, które nie są używane, często są usypane i ponownie uruchamiane dopiero po wykonaniu do nich nowego żądania. W przypadku płatnych planów większości platform hostingowych jest znacznie bardziej responsywna.

## Dopracowanie wdrożenia projektu w Platform.sh

W tej sekcji zajmiemy się dopracowaniem wdrożenia przez utworzenie superuzytkownika, podobnie jak miało to miejsce w środowisku lokalnym. Ponadto zapewnimy projektowi większe bezpieczeństwo przez przypisanie opcji DEBUG wartości False, aby użytkownicy nie otrzymywali żadnych informacji dodatkowych w komunikatach błędów, ponieważ tego rodzaju dane mogą ułatwić przeprowadzenie ataku na serwer.

## Utworzenie superużytkownika w Platform.sh

Baza danych dla projektu na zdalnym serwerze została skonfigurowana, ale jest zupełnie pusta. Wszyscy utworzeni wcześniej użytkownicy istnieją jedynie w lokalnej wersji projektu.

W celu utworzenia superużytkownika we wdrożonej wersji projektu trzeba uruchomić sesję SSH (bezpieczna powłoka), za pomocą której będzie można wykonywać polecenia administracyjne na zdalnym serwerze.

(11 env)learning log\$ platform environment:ssh

— \ — | — / — | — ( - ' \ | , \ / | | | ( ) / ||

```
Welcome to Platform.sh.
```

```
web@11_project.0:~$ ls ①
accounts learning_logs 11_project logs manage.py requirements.txt
    requirements_remote.txt static
web@11_project.0:~$ python manage.py createsuperuser ②
Username (leave blank to use 'web'): 11_admin_live ③
Email address:
Password:
Password (again):
Superuser created successfully.
web@11_project.0:~$ exit ④
logout
Connection to ssh.us-3.platform.sh closed.
(11_env)learning_log$ ⑤
```

---

W trakcie pierwszego wykonania polecenia `platform:environment:ssh` możesz otrzymać inny komunikat dotyczący uwierzytelnienia w hoście. Jeżeli zobacysz pokazany tutaj komunikat, wpisz Y i naciśnij klawisz *Enter*. W ten sposób powinieneś zostać zalogowany w zdalnej sesji powłoki.

Po wydaniu polecenia `ssh` powłoka działa dokładnie jak zwykła powłoka na zdalnym serwerze. Zwróć uwagę na zmianę znaku zachęty, który teraz wskazuje na sesję `web` powiązaną z projektem `11_project` (patrz wiersz ①). Jeżeli wydasz polecenie `ls`, sprawdzisz, które pliki zostały przekazane na serwer `Platform.sh`.

Wydaj to samo polecenie `python manage.py createsuperuser` (patrz wiersz ②), które znasz z rozdziału 18. Tym razem jako nazwę administratora podajemy `11_admin_live` (patrz wiersz ③), aby odróżniała się od użytkownika w lokalnej wersji projektu. Po zakończeniu pracy ze zdalną sesją powłoki należy wydać polecenie `exit` (patrz wiersz ④). Znak zachęty powłoki będzie wskazywał, że ponownie znajdujesz się w systemie lokalnym (patrz wiersz ⑤).

Teraz można już na końcu adresu URL wdrożonej aplikacji dodać `/admin/` i zalogować się do witryny administracyjnej. Jeżeli inni użytkownicy zaczęli korzystać z projektu *Learning Log*, to pamiętaj, że masz dostęp do wszystkich wprowadzonych przez nich danych. Nie zawiedź ich, a nadal będą mieli do Ciebie zaufanie i będą dodawali następne wpisy.

**UWAGA** *Użytkownicy Windows będą w tym miejscu używać tych samych poleceń (czyli `ls` zamiast `dir`), ponieważ pracujemy w powłoce systemu Linux za pomocą zdalonego połączenia.*

## Zabezpieczenie wdrożonego projektu

Jednym z poważnych problemów bezpieczeństwa, który istnieje w aktualnie wdrożonym przez nas projekcie, jest ustawienie `DEBUG=True` w pliku `settings.py`. Ustawienie tej opcji powoduje wyświetlenie komunikatów debugowania w przypadku wystąpienia błędu. Strony błędów Django mogą dostarczyć naprawdę użytecznych informacji debugowania podczas pracy nad projektem, ale informacje

te zdecydowanie będą ujawniać zbyt wiele potencjalnym atakującym, jeśli nadal będą wyświetlane we wdrożonej aplikacji.

Aby przekonać się, jaki to może być problem, przejdź na stronę główną wdrożonego projektu. Zaloguj się do konta użytkownika i dodaj `/topics/999/` na końcu adresu URL strony głównej. Przyjmując założenie, że nie utworzono jeszcze tysiąca tematów, powinien zostać wyświetlony komunikat zawierający błąd `DoesNotExist at /topics/999/`. Po przewinięciu tekstu w dół zobaczysz wiele informacji dotyczących projektu i serwera. Nie chcesz ujawniać tych informacji użytkownikom i zdecydowanie nie chcesz, aby trafiły one do kogoś, kto chciałby przeprowadzić atak na Twoją witrynę.

Można zapobiec wyświetaniu tego rodzaju informacji przez wdrożony projekt. W tym celu wystarczy ustawienie `DEBUG = False` w części pliku `settings.py`, która ma zastosowanie dla wdrożonego projektu. Dzięki temu nadal będziesz otrzymywać informacje debugowania dla aplikacji działającej lokalnie, ponieważ są one wówczas niezwykle użyteczne. Natomiast nie będą wyświetlane w przypadku aplikacji wdrożonej na zdalnym serwerze.

W edytorze tekstu otwórz teraz plik `settings.py` i dodaj wiersz kodu do sekcji dotyczącej wdrożonej aplikacji na serwerze Platform.sh.

*Plik settings.py:*

---

```
--cięcie--  
if config.is_valid_platform():  
    ALLOWED_HOSTS.append('.platformsh.site')  
    DEBUG = False  
--cięcie--
```

---

Cała praca związana z konfiguracją wdrożonej na serwerze Platform.sh wersji projektu opłaciła się. Kiedy chcemy dostosować tę wersję projektu, wystarczy po prostu zmienić odpowiednią sekcję przygotowanej wcześniej konfiguracji.

## Zatwierdzenie zmian i przekazanie ich do serwera

Konieczne jest zatwierdzenie w systemie Git zmian wprowadzonych w pliku `settings.py`, a następnie przekazanie ich do Platform.sh. Poniżej przedstawiłem sesję terminala pokazującą ten proces:

---

```
(11_env)learning_log$ git commit -am "Ustawienie wartości False opcji DEBUG  
we wdrożonej aplikacji." ①  
[main d2ad0f7] Ustawienie wartości False opcji DEBUG we wdrożonej aplikacji.  
1 file changed, 1 insertions(+)  
(11_env)learning_log$ git status ②  
On branch main  
nothing to commit, working directory clean  
(11_env)learning_log$
```

---

W wierszu ① wydajemy polecenie `git commit` wraz z krótkim, choć jasnym komunikatem operacji zatwierdzenia zmian. Pamiętaj, że opcja `-am` spowoduje zatwierdzenie przez Gita wszystkich zmodyfikowanych plików oraz zarejestrowanie podanego komunikatu. Git rozpoznaje, że zmieniony został tylko jeden plik, i zatwierdza tę zmianę w repozytorium.

W wierszu ② komunikat stanu wskazuje, że pracujemy w gałęzi *main* repozytorium i nie ma żadnych nowych zmian do zatwierdzenia. Sprawdzenie stanu repozytorium jest bardzo ważne, zanim przekażemy zmiany na zdalny serwer. Jeżeli nie widzisz takiego komunikatu, pewne zmiany mogły nie zostać zatwierdzone, więc nie będą przekazane do serwera. Możesz spróbować ponownie wydać polecenie `commit`, ale jeśli nie wiesz, jak rozwiązać problem, zajrzyj do dodatku D, w którym znajdziesz więcej informacji na temat pracy z systemem kontroli wersji Git.

Teraz uaktualnione repozytorium przekazujemy do Platform.sh:

---

```
(11_env)learning_log$ platform push
Are you sure you want to push to the main (production) branch? [Y/n] Y
Pushing HEAD to the existing environment main
--cięcie--
To git.us-3.platform.sh:wmye2fx7wwqgu.git
  fce0206..d2ad0f7  HEAD -> main
(11_env)learning_log$
```

---

Platform.sh rozpoznaje, że repozytorium zostało uaktualnione. Przebudowuje więc projekt, aby zagwarantować uwzględnienie wszystkich wprowadzonych w nim zmian. Baza danych nie zostanie przebudowana, więc nie utracisz żadnych danych.

Aby sprawdzić, czy zmiana została skutecznie wprowadzona, ponownie przejdź pod adres URL kończący się na `/topics/999/`. Tym razem otrzymasz komunikat *Server Error (500)* i nie zostaną ujawnione żadne informacje wrażliwe dotyczące projektu.

## Utworzenie własnych stron błędu

W rozdziale 19. skonfigurowaliśmy aplikację *Learning Log* do zgłoszenia błędu o kodzie 404, gdy użytkownik zażąda tematu, którego nie jest właścicielem. Natomiast przed chwilą spotkaliśmy się z błędem o kodzie 500 (wewnętrzny błąd serwera). Błąd o kodzie 404 zwykle oznacza, że kod Django jest prawidłowy, a żądany obiekt nie istnieje. Z kolei błąd o kodzie stanu 500 zwykle oznacza istnienie błędu w utworzonym przez Ciebie kodzie, na przykład w funkcji zdefiniowanej w pliku `views.py`. Obecnie Django generuje tę samą ogólną stronę błędu w obu wymienionych sytuacjach. Jednak istnieje możliwość przygotowania własnych szablonów dla stron błędów o kodach 404 i 500. Utworzmy je i dopasujemy ich wygląd do ogólnego wyglądu aplikacji *Learning Log*. Przedstawione tutaj szablony muszą się znaleźć w katalogu głównym szablonów.

## Utworzenie własnych szablonów

W katalogu *learning\_log* utwórz nowy podkatalog o nazwie *templates*. Następnie w nowym katalogu utwórz plik *404.html*. Ścieżka dostępu do tego pliku powinna mieć postać *learning\_log/templates/404.html*. Teraz w tym nowym pliku umieść poniższy fragment kodu.

*Plik 404.html:*

---

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
    <h2>Żądany element nie istnieje. (404)</h2>
{% endblock page_header %}
```

---

Ten prosty szablon zapewnia wyświetlenie ogólnego komunikatu na stronie błędu 404, a jego styl jest dopasowany do stylu całej witryny internetowej.

Utwórz następny plik, tym razem o nazwie *500.html*, i umieść w nim poniższy fragment kodu.

*Plik 500.html:*

---

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
    <h2>Wystąpił wewnętrzny błąd serwera. (500)</h2>
{% endblock page_header %}
```

---

Aby móc wykorzystać przygotowane powyżej szablony dla stron błędów, musimy wprowadzić małą zmianę w pliku *settings.py*.

*Plik settings.py:*

---

```
--cięcie--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'],
        'APP_DIRS': True,
        '--cięcie--'
    },
]
--cięcie--
```

---

Powyższa zmiana nakazuje Django sprawdzenie katalogu głównego szablonów podczas szukania szablonów dla generowanych stron oraz wszelkich innych szablonów niepowiązanych z określona aplikacją.

## Przekazywanie zmian do Platform.sh

Teraz możemy już zatwierdzić zmiany wprowadzone w szablonach oraz przekazać je do Platform.sh:

```
(11_env)learning_log$ git add . ①
(11_env)learning_log$ git commit -am "Dodanie własnych stron błędów 404 i 500." ②
 3 files changed, 11 insertions(+), 1 deletion(-)
  create mode 100644 templates/404.html
  create mode 100644 templates/500.html
(11_env)learning_log$ platform push ③
--cięcie--
remote: Verifying deploy.... done.
To git.us-3.platform.sh:wmye2fx7wwqgu.git
  d2ad0f7..9f042ef  HEAD -> main
(11_env)learning_log$
```

W wierszu ① wydaliśmy polecenie `git add .` (zwrócić uwagę na kropkę na końcu polecenia), ponieważ w projekcie zostały utworzone nowe pliki. W ten sposób nakażaliśmy systemowi Git ich monitorowanie. Następnie zatwierdziliśmy wprowadzone zmiany (patrz wiersz ②) i uaktualniony projekt przekazaliśmy do Platform.sh (patrz wiersz ③).

Od teraz wyświetlane strony błędów powinny mieć nadane takie same style jak pozostała część witryny internetowej, co na pewno będzie korzystne z punktu widzenia użytkownika.

## Nieustanna rozbudowa

Po wdrożeniu aplikacji *Learning Log* w serwerze WWW możesz ją nadal rozbudowywać lub opracować własne projekty przeznaczone do wdrożenia. Proces uaktualniania projektów jest całkiem spójny.

Przede wszystkim wszelkie zmiany trzeba wprowadzić w projekcie lokalnym. Jeżeli zmiany wiążą się z powstaniem nowych plików, te pliki należy dodać do repozytorium Git za pomocą polecenia `git add .` (upewnij się o umieszczeniu kropki na końcu polecenia). Wszelkie zmiany wymagające przeprowadzenia migracji bazy danych będą wymagały wydania powyższego polecenia, ponieważ każda migracja generuje nowy plik migracji.

Kolejnym krokiem jest zatwierdzenie zmian w repozytorium za pomocą polecenia `git commit -am "Komunikat dotyczący zatwierdzenia"`. Następnie zmiany można przekazać do Platform.sh, co wymaga wydania polecenia `platform push`. Teraz przejdź do wdrożonej aplikacji i upewnij się, że wprowadzone zmiany przyniosły oczekiwany efekt.

Bardzo łatwo można popełnić błąd w przedstawionym procesie, więc nie bądź zaskoczony, gdy coś pójdzie źle. Jeżeli kod nie działa, przejrzyj wprowadzone zmiany i spróbuj znaleźć błąd. Natomiast jeśli nie jesteś w stanie wychwycić błędu lub nie wiesz, jak wycofać te zmiany, zajrzyj do dodatku C, w którym znajdziesz wiele przydatnych sugestii dotyczących uzyskiwania pomocy. Nie wstydz

się prosić o pomoc — każdy kiedyś uczył się budować projekty i mógł zadawać pytania, które teraz Ty chcesz zadać. Istnieje więc spore prawdopodobieństwo, że ktoś będzie zadowolony, jeśli będzie mógł Ci pomóc. Dzięki rozwiązywaniu pojawiających się problemów zdobywasz większe umiejętności, aż w pewnym momencie zacznesz tworzyć konkretne, niezawodne projekty i sam będziesz odpowiadać na pytania zadawane przez innych.

## Usunięcie projektu z Platform.sh

Doskonałym ćwiczeniem będzie kilkukrotne przejście przez proces wdrożenia tego samego projektu lub serii małych, aby nabyć większego doświadczenia we wdrażaniu aplikacji. Warto jednak wiedzieć, jak można usunąć wdrożony projekt. Z jednej strony Platform.sh ogranicza liczbę projektów obsługiwanych przez bezpłatne konto, a z drugiej strony Ty też możesz nie chcieć zaśmiecać konta przykładowymi projektami służącymi jedynie celom treningowym.

Projekt możesz usunąć za pomocą polecenia powłoki:

---

```
(11_env)learning_log$ platform project:delete
```

---

Trzeba będzie potwierdzić przeprowadzenie tej destrukcyjnej operacji. Udziel odpowiedzi na wyświetlane pytania, a projekt zostanie usunięty.

Polecenie `platform create` przekazuje do lokalnego repozytorium Git odniesienie do zdalnego repozytorium na serwerach Platform.sh. To odniesienie można usunąć także za pomocą polecenia powłoki:

---

```
(11_env)learning_log$ git remote
platform
(11_env)learning_log$ git remote remove platform
```

---

Polecenie `git remote` wyświetla listę nazw wszystkich zdalnych adresów URL powiązanych z bieżącym repozytorium. Polecenie `git remote remove nazwa_zdalna` powoduje, że z repozytorium lokalnego zostaje usunięty wskazany zdalny adres URL.

Zasoby projektu można również usunąć przez zalogowanie się do witryny Platform.sh i przejście do panelu sterowania pod adresem <https://console.platform.sh>. Na tej stronie znajduje się lista wszystkich aktywnych projektów. Kliknij trzy kropki w menu projektu, a następnie wybierz opcję *Edit Plan*. Pojawi się strona z cennikiem dla danego projektu. Kliknij przycisk *Delete Project* u dołu strony, a przejdź na stronę potwierdzenia operacji usunięcia projektu. Nawet jeśli projekt został usunięty z poziomu powłoki, i tak warto zapoznać się z panelem sterownika dostawcy usług hostingu.

**UWAGA** Usunięcie projektu w Platform.sh nie ma nic wspólnego z wersją aplikacji w środowisku lokalnym. Jeżeli nikt nie używał jeszcze wdrożonej przez Ciebie aplikacji i jedynie ćwiczył proces wdrażania projektu, całkiem rozsądne jest jego usunięcie z Platform.sh, a następnie przeprowadzenie ponownego wdrożenia. Pamiętaj, że jeśli coś przestanie działać, to być może napotkasz ograniczenia dotyczące konta bezpłatnego dostawcy usług hostingu.

## ZRÓB TO SAM

**20.3. Wdrożenie bloga.** Przeprowadź wdrożenie w Platform.sh projektu bloga, nad którym pracujesz. Aby zapewnić wdrożonej aplikacji względne bezpieczeństwo, upewnij się, że przypisałeś opcji DEBUG wartość False, aby w przypadku problemów użytkownicy nie widzieli wyświetlanej pełnej strony błędu Django.

**20.4. Rozbudowa aplikacji *Learning Log*.** Dodaj jedną funkcję do aplikacji *Learning Log*, a następnie przekaż zmiany do serwera produkcyjnego. Spróbuj wprowadzić niewielką zmianę, na przykład nieco obszerniejszy opis przeznaczenia projektu wyświetlany na stronie głównej. Następnie postaraj się dodać trochę bardziej zaawansowaną funkcję, na przykład możliwość publicznego udostępnienia tematu przez jego właściciela. Taka zmiana będzie wymagała użycia atrybutu o nazwie public jako części modelu Topic (wartością domyślną wymienionego atrybutu powinno być False) oraz elementu formularza na stronie new\_topic pozwalającego użytkownikowi na oznaczenie tematu jako prywatnego lub publicznego. Następnie trzeba będzie przeprowadzić migrację projektu i zmodyfikować kod w pliku views.py, aby tematy publiczne mogły być wyświetlane również przez nieuwierzytelnionych użytkowników.

# Podsumowanie

W tym rozdziale dowiedziałeś się, jak nadać projektom minimalistyczny, choć elegancki i profesjonalny, wygląd za pomocą biblioteki Bootstrap oraz aplikacji django-[bootstrap5](#). Użycie biblioteki Bootstrap zapewnia dostęp do stylów działających spójnie na praktycznie każdym urządzeniu, na którym użytkownicy mogą korzystać z Twojej aplikacji.

Poznałeś szablony Bootstrapa i zobaczyłeś, jak użyć szablonu *Navbar static* do przygotowania elegancko wyglądającej aplikacji *Learning Log*. Nauczyłeś się wykorzystywać element Jumbotron do utworzenia wyróżniającej się treści strony głównej, a także dowiedziałeś się, jak spójnie nadawać styl wszystkim stronom witryny internetowej.

W ostatniej części projektu zajęliśmy się jego wdrożeniem na zdalnych serwerach, aby przygotowana wcześniej aplikacja stała się dostępna dla każdego. Utworzyłeś konto w serwisie Platform.sh i zainstalowałeś pewne narzędzia pomagające w zarządzaniu procesem wdrażania. Systemu kontroli wersji

Git użyłeś do zatwierdzenia działającego projektu, umieszczenia go w repozytorium, a następnie do przekazania tego repozytorium do serwerów Platform.sh. Na koniec dowiedziałeś się, jak zwiększyć bezpieczeństwo aplikacji dzięki przypisaniu wartości `Fals` se opcji `DEBUG` dla projektu wdrożonego w serwerze produkcyjnym. Przygotowałeś także własne strony błędów, aby zapewnić elegancką obsługę nieuniknionych błędów, które się pojawią.

Skoro zakończyłeś pracę nad aplikacją *Learning Log*, możesz rozpocząć tworzenie własnych projektów. Rozpocznij od prostych zadań, a zanim zaczniesz zwiększać poziom skomplikowania projektu, upewnij się, że jak na razie wszystko działa prawidłowo. Życzę Ci przyjemnej nauki oraz powodzenia we własnych projektach!

# A

## Instalacja Pythona i rozwiązywanie problemów



PYTHON JEST DOSTĘPNY W KILKU RÓŻNYCH WERSJACH I MOŻNA GO ZAINSTALOWAĆ NA WIELE RÓŻNYCH SPOSOBÓW W KAŻDYM SYSTEMIE OPERACYJNYM. TEN DODATEK OKAŻE SIĘ UŻYTECZNY W PRZYPADKU, GDY PROCEDURA instalacji przedstawiona w rozdziale 1. nie zadziała lub chcesz zainstalować inne wydanie Pythona niż standardowo dostarczane z używanym przez Ciebie systemem operacyjnym.

### Python w Windows

Procedura przedstawiona w rozdziale 1. pokazuje instalację Pythona za pomocą oficjalnego programu instalacyjnego, który pobrałeś z witryny <https://www.python.org/>. Jeżeli po jej przeprowadzeniu nie możesz uruchomić Pythona, informacje zamieszczone w tym podrozdziale powinny Ci pomóc.

## Użycie polecenia py zamiast python

Jeżeli uruchomisz w miarę nową wersję programu instalacyjnego Pythona, a następnie w wierszu poleceń wydasz polecenie `python`, powinieneś zobaczyć wyświetlony znak zachęty `>>>`. Gdy Windows nie rozpoznaje polecenia `python`, wówczas uruchamia aplikację sklepu Microsoft Store, sądząc, że Python nie został zainstalowany, bądź powoduje wyświetlenie komunikatu błędu informującego o nieznalezieniu Pythona. Jeżeli nastąpiło przejście do aplikacji sklepu Microsoft Store, zamknij ją, ponieważ znacznie lepszym rozwiązaniem będzie użycie oficjalnego programu instalacyjnego Pythona z witryny <https://python.org/> niż wersji dostarczanej przez Microsoft.

Najprostszym rozwiązaniem, niewymagającym wprowadzenia żadnych zmian w systemie, jest użycie polecenia `py`. Jest to narzędzie Windowsa umożliwiające znalezienie najnowszej wersji Pythona zainstalowanej w systemie oraz uruchomienie tego interpretera. Jeżeli to polecenie działa, wystarczy go używać wszędzie tam, gdzie w książce pojawia się polecenie `python` lub `python3`.

## Ponowna instalacja Pythona

Najczęstszym powodem niedziałania polecenia `python` jest niezaznaczenie opcji *Add python.exe to PATH* podczas instalacji Pythona. Ten błąd łatwo popełnić. Zmienna systemowa `PATH` to ustawienie systemowe wskazujące Pythonowi, gdzie ma szukać najczęściej używanych programów. Jeżeli zapomnisz o tej opcji, Windows nie będzie wiedział, gdzie znajduje się interpreter Pythona.

Najprostszym rozwiązaniem jest ponowne uruchomienie programu instalacyjnego. Jeżeli w witrynie <https://python.org/> znajduje się nowsza wersja, pobierz ją i uruchom. Tym razem upewnij się, że jest zaznaczone pole wyboru *Add python.exe to PATH*.

Jeżeli masz już najnowszą wersję programu instalacyjnego, uruchom go ponownie i wybierz opcję *Modify*. Zobaczysz listę funkcjonalności opcjonalnych. Zachowaj opcje domyślne widoczne na ekranie. Kliknij przycisk *Dalej* i zaznacz pole wyboru *Add Python to Environment Variables*. Następnie kliknij przycisk *Zainstaluj*. Program instalacyjny rozpozna, że Python jest już zainstalowany, i doda do zmiennej systemowej `PATH` położenie interpretera. Upewnij się, że zostały zamknięte wszystkie okna wiersza polecenia, a następnie ponownie wydaj polecenie `python`. Na ekranie powinien zostać wyświetlony znak zachęty Pythona, czyli `>>>`.

## Python w systemie macOS

Procedura przedstawiona w rozdziale 1. pokazuje instalację Pythona za pomocą oficjalnego programu instalacyjnego, który pobrałeś z witryny <https://www.python.org/>. Oficjalny program instalacyjny sprawdza się doskonale od wielu lat, choć mogą się zdarzyć sytuacje, w których coś pójdzie nie tak. Ten podrozdział może okazać się przydatny, jeżeli wspomniane podejście nie działa zgodnie z oczekiwaniami.

## Przypadkowa instalacja wersji dostarczanej przez Apple

Jeżeli wydasz polecenie `python3`, a Python nie został jeszcze zainstalowany w systemie, prawdopodobnie otrzymasz komunikat wspominający o konieczności zainstalowania *narzędzi programistycznych działających w powłoce*. Wówczas najlepszym rozwiązaniem będzie zamknięcie okna wyświetlającego taki komunikat, przejście do witryny <https://python.org/>, pobranie znajdującego się tam programu instalacyjnego i jego uruchomienie.

Jeżeli zdecydujesz się na zainstalowanie *narzędzi programistycznych działających w powłoce*, razem z tymi narzędziami otrzymasz dostarczaną przez Apple wersję Pythona. Jedyny związek z nią problem polega na tym, że zwykle jest ona nieco starsza niż najnowsza dostępna oficjalna wersja Pythona. Jednak nadal możesz pobrać i uruchomić oficjalny program instalacyjny z witryny <https://python.org/>, a wówczas polecenie `python3` będzie prowadziło do nowej wersji. Nie przejmuj się koniecznością instalacji narzędzi programistycznych, znajdziesz w nich wiele użytecznych narzędzi, w tym również omówiony w dodatku D system kontroli wersji Git.

## Python 2 w starszych wydaniach systemu macOS

W starszych wydaniach systemu macOS, sprzed wersji Monterey (macOS 12), domyślnie była zainstalowana przestarzała wersja Pythona 2. W tych systemach polecenie `python` prowadziło do tego przestarzałego interpretera. Jeżeli używasz systemu macOS z zainstalowanym wydaniem Pythona 2, upewnij się o używaniu polecenia `python3` — wówczas zawsze będziesz korzystać z samodzielnie zainstalowanej wersji Pythona.

## Python w systemie Linux

Python jest zainstalowany domyślnie niemal w każdym systemie Linux. Jeżeli wersja domyślna jest starsza niż 3.9, powinieneś zainstalować najnowszą dostępną. Najnowszą wersję możesz również zainstalować, jeśli chcesz uzyskać dostęp do najnowszej funkcjonalności wprowadzonej do języka, np. poprawionych komunikatów błędów. Przedstawione tutaj informacje sprawdzą się w większości dystrybucji wykorzystujących menedżer pakietów apt.

## Używanie domyślnej instalacji Pythona

Jeżeli chcesz użyć wersji Pythona wskazywanej przez polecenie `python3`, upewnij się, że masz zainstalowane następujące trzy pakiety dodatkowe:

---

```
$ sudo apt install python3-dev python3-pip python3-venv
```

---

Te pakiety zawierają narzędzia użyteczne dla programistów oraz narzędzia umożliwiające na instalację pakietów zewnętrznych, takich jak wykorzystane w rozdziałach poświęconych projektom praktycznym.

## Instalacja najnowszej wersji Pythona

W tym miejscu użyjemy pakietu deadsnakes, który ułatwi instalację wielu wersji Pythona. W powłoce wydaj następujące polecenia:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa  
$ sudo apt update  
$ sudo apt install python3.11
```

Powyższe polecenia powodują zainstalowanie w systemie Pythona w wersji 3.11.

Jeżeli teraz wydasz poniższe polecenie, uruchomisz w powłoce sesję Pythona 3.11:

```
$ python3.11  
>>>
```

Każde polecenie `python` w książce zastąp poleceniem `python3.11`. To polecenie będzie stosowane też podczas uruchamiania programów Pythona z poziomu powłoki.

Musisz jeszcze zainstalować dwa kolejne pakiety, aby maksymalnie wykorzystać instalację Pythona.

```
$ sudo apt install python3.11-dev python3.11-venv
```

Ta pakiety obejmują moduły, które będą potrzebne podczas instalowania i uruchamiania pakietów zewnętrznych, takich jak wykorzystane w rozdziałach poświęconych projektom praktycznym w części drugiej książki.

**UWAGA** *Pakiet deadsnakes jest aktywnie rozwijany już od wielu lat. Po pojawienniu się nowszej wersji Pythona można nadal używać tych samych poleceń, zastępując `python3.11` aktualnie najnowszą dostępną wersją.*

## Sprawdzenie aktualnie używanej wersji Pythona

Jeżeli masz problemy z uruchamianiem Pythona bądź instalowaniem pakietów dodatkowych, pomocne może się okazać ustalenie, z której dokładnie wersji Pythona korzystasz. W systemie możesz mieć zainstalowanych wiele wersji Pythona i wówczas niekoniecznie będzie jasne, która z nich jest aktualnie używana.

W powłoce wydaj następujące polecenie:

---

```
$ python --version  
Python 3.11.0
```

---

W ten sposób dokładnie ustalisz, do której wersji Pythona prowadzi polecenie `python`. Krótsza wersja tego polecenia, `python -V`, powoduje wygenerowanie tych samych danych wyjściowych.

## Słowa kluczowe Pythona i wbudowane funkcje

Python zawiera pewien zbiór słów kluczowych oraz wbudowanych funkcji. Trzeba koniecznie o nich pamiętać podczas nadawania nazw zmiennym. W nazwach zmiennych nie wolno używać żadnych słów kluczowych Pythona lub wbudowanych funkcji. W przeciwnym razie po prostu je nadpiszesz.

W tym podrozdziale przedstawię wszystkie słowa kluczowe Pythona i nazwy wbudowanych funkcji. Dzięki temu będziesz wiedział, jakich nazw unikać podczas tworzenia zmiennych.

### Słowa kluczowe Pythona

Każde z wymienionych poniżej słów kluczowych ma konkretne znaczenie. Jeżeli spróbujesz użyć tych słów w charakterze nazwy zmiennej, otrzymasz komunikat błędu.

---

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

---

### Wbudowane funkcje Pythona

Kiedy spróbujesz użyć dowolnej z wymienionych poniżej nazw w charakterze nazwy zmiennej, wówczas nie otrzymasz komunikatu błędu, ale zamiast tego po prostu spowodujesz nadpisanie zachowania danej funkcji.

abs()	complex()	hash()	min()	slice()
aiter()	delattr()	help()	next()	sorted()
all()	dict()	hex()	object()	staticmethod()
any()	dir()	id()	oct()	str()
anext()	divmod()	input()	open()	sum()
ascii()	enumerate()	int()	ord()	super()
bin()	eval()	isinstance()	pow()	tuple()
bool()	exec()	issubclass()	print()	type()
breakpoint()	filter()	iter()	property()	vars()
bytearray()	float()	len()	range()	zip()
bytes()	format()	list()	repr()	<u>import</u> ()
callable()	frozenset()	locals()	reversed()	
chr()	getattr()	map()	round()	
classmethod()	globals()	max()	set()	
compile()	hasattr()	memoryview()	setattr()	

# B

## Edytory tekstu i środowiska IDE



PROGRAMIŚCI SPĘDZAJĄ DUŻO CZASU NA TWORZENIU, CZYTANIU I EDYTOWANIU KODU ŹRÓDŁOWEGO. DLATEGO UŻYWANIE DO TEGO CELU EDYTORA TEKSTU LUB ZINTEGROWANEGO ŚRODOWISKA PROGRAMISTYCZ-

nego, które zapewni im maksymalną możliwą efektywność, ma bardzo duże znaczenie. Dobry edytor tekstu będzie wykonywał proste zadania, takie jak kolorowanie składni, które podkreślają strukturę kodu, aby ułatwiać programistom wyłapywanie najczęściej popełnianych błędów podczas pracy. Jednocześnie dobry edytor nie powinien rozpraszać programisty i odciągać go od myślenia nad kodem. Ponadto dobry edytor tekstu powinien również obsługiwać automatyczne wcięcia, znaczniki pokazujące odpowiednią długość wiersza oraz mieć skróty klawiszowe dla często wykonywanych operacji.

Zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*, IDE) to edytor tekstu powiązany z wieloma innymi narzędziami, np. interaktywnym debuggerem i narzędziem służącym do introspekcji kodu źródłowego. Środowisko IDE analizuje kod podczas jego wprowadzania i próbuje zebrać jak najwięcej informacji o tworzonym projekcie. Na przykład gdy zaczniesz wpisywać nazwę funkcji, IDE może wyświetlić wszystkie akceptowane przez nią argumenty. Takie rozwiązanie okazuje się niezwykle użyteczne, gdy wszystko działa, a Ty rozumiesz to, co widzisz na ekranie. Jednocześnie środowisko IDE może być przytaczające dla początkującego programisty i utrudniać rozwiązywanie problemów, gdy nie potrafi on ustalić, dlaczego dany kod nie działa w IDE.

Obecnie zacierają się granice między edytorami tekstu i środowiskami IDE. Większość popularnych edytorów ma jakąś funkcjonalność, którą kiedyś oferowały

wyłącznie IDE. Podobnie większość środowisk IDE można skonfigurować do działania w lekkim trybie, mniej rozpraszającym podczas pracy, a jednocześnie pozwalającym korzystać z bardziej zaawansowanych funkcji, gdy okażą się potrzebne.

Jeżeli masz już zainstalowany edytor lub środowisko IDE, które lubisz, i jeśli jest on skonfigurowany do pracy z najnowszą wersją Pythona zainstalowaną w Twoim systemie, zachęcam Cię do pozostawienia takiej konfiguracji. Poznawanie różnych edytorów może być fascynujące, choć jednocześnie będzie odciągało od poznawania nowego języka programowania.

Natomiast jeśli jeszcze nie masz zainstalowanego edytora bądź środowiska IDE, z wielu powodów zachęcam do wypróbowania VS Code:

- Jest to produkt bezpłatny i dostępny na licencji open source.
- VS Code można zainstalować w większości systemów operacyjnych.
- Jest to produkt przyjazny dla początkujących i jednocześnie oferujący na tyle potężne możliwości, że stanowi podstawowy edytor dla wielu zawodowych programistów.
- Znajduje zainstalowane wersje Pythona i zwykle nie wymaga żadnej konfiguracji podczas uruchamiania pierwszych programów w Pythonie.
- Ma zintegrowany terminal, więc dane wyjściowe programu pojawiają się w tym samym oknie, w którym tworzysz kod.
- Dostępne jest rozszerzenie o nazwie *Python*, dzięki któremu ten edytor okazuje się niezwykle efektywny podczas tworzenia kodu źródłowego w Pythonie i jego późniejszej obsługi.
- Ma potężne możliwości w zakresie dostosowywania do własnych potrzeb, więc można go dowolnie skonfigurować pod kątem ulubionego sposobu pracy z kodem źródłowym.

W tym dodatku zajmiesz się przygotowaniem konfiguracji edytora tekstu VS Code, która pomoże Ci w efektywniejszej pracy. Poznasz również wybrane skróty klawiszowe, które umożliwiają jeszcze bardziej efektywną pracę. Umiejętność szybkiego pisania nie ma aż tak ogromnej wagę w programowaniu, jak wiele osób może sądzić. Natomiast użyteczne jest poznanie stosowanego edytora oraz umiejętności efektywnej pracy z nim.

Mimo to VS Code nie jest produktem dla każdego. Jeżeli nie sprawdza się w Twoim systemie bądź za bardzo Cię rozprasza, istnieje jeszcze wiele innych interesujących edytorów. Dlatego w tym dodatku pokróćce przedstawię także wiele alternatywnych edytorów i środowisk IDE wartych uwagi.

## Efektywna praca z VS Code

W rozdziale 1. zainstalowałeś VS Code i dodałeś rozszerzenie *Python*. Teraz przeprowadzisz konfigurację dodatkową oraz poznasz wybrane skróty klawiszowe, aby móc efektywnie pracować z kodem źródłowym za pomocą tego edytora.

## Konfigurowanie VS Code

Istnieje kilka sposobów na zmianę domyślnych ustawień konfiguracyjnych VS Code. Część z nich można wprowadzić za pomocą interfejsu użytkownika, podczas gdy inne będą wymagały modyfikacji plików konfiguracyjnych. Te zmiany czasami wpływają na wszystko, co będziesz robić za pomocą VS Code, inne zaś wpływają jedynie na pliki znajdujące się w katalogu zawierającym dany plik konfiguracyjny.

Na przykład jeśli plik konfiguracyjny znajdzie się w katalogu *projekty\_pythona*, to zdefiniowane w nim ustawienia będą wpływać jedynie na pliki w tym katalogu (i jego podkatalogach). To jest świetna funkcjonalność, ponieważ pozwala przygotowywać ustawienia dla danego projektu, nadpisując ustawienia globalne.

## Używanie tabulatorów i spacji

Jeżeli w kodzie źródłowym będziesz łączyć tabulatory i spacje, może to doprowadzić do powstania błędów, których wykrycie okaże się niezwykle trudne. Podczas pracy w pliku *.py*, jeśli w VS Code jest zainstalowane rozszerzenie *Python*, edytor będzie skonfigurowany w taki sposób, że naciśnięcie klawisza *Tab* spowoduje wstawienie czterech spacji. Jeżeli jednak tworzysz własny kod i masz zainstalowane rozszerzenie *Python* w VS Code, prawdopodobnie nigdy nie będziesz mieć problemów z tabulatorami i spacjami.

Jednak instalacja VS Code może nie być poprawnie skonfigurowana. Ponadto w pewnym momencie możesz pracować z plikiem, w którym użyto jedynie tabulatorów bądź połączenia tabulatorów i spacji. Gdy podejrzewasz problem z tabulatorami i spacjami, spójrz na pasek stanu na dole okna VS Code i kliknij *Spaces* lub *Tab Size*. Pojawi się rozwijane menu umożliwiające przełączanie się między używaniem tabulatorów i spacji. Ponadto można wybrać domyślny poziom wcięć oraz skonwertować wszystkie wcięcia w pliku na tabulatory bądź spacje.

Jeżeli analizując kod utworzony przez kogoś innego, nie masz pewności, w jaki sposób zostały utworzone wcięcia, tabulatory lub spacje, to zaznacz kilka wierszy kodu, a wówczas niewidoczne białe znaki staną się widoczne. Spacja jest przedstawiana za pomocą kropki, tabulator zaś za pomocą strzałki.

**UWAGA** *W programowaniu preferowane jest używanie spacji zamiast tabulatorów, ponieważ spacje są bezproblemowo interpretowane przez wszystkie narzędzia pracujące z plikami kodu źródłowego. Z kolei wielkość tabulatora może być interpretowana odmiennie przez poszczególne narzędzia, co prowadzi do błędów, których wychwycenie może okazać się niezwykle trudne.*

## Zmiana motywu koloru

VS Code domyślnie używa ciemnego motywu. Jeżeli chcesz to zmienić, kliknij menu *File* (w systemie macOS kliknij menu *Code*), następnie *Preferences* i wybierz *Color Theme*. Na ekranie zostanie wyświetlona rozwijana lista, za pomocą której można wybrać motyw koloru.

## Ustawianie wskaźnika długości linii

Większość edytorów pozwala zdefiniować wizualny wskaźnik, najczęściej w postaci pionowej linii wskazującej miejsce, w którym powinien zakończyć się wiersz kodu. W społeczności Pythona przyjęła się konwencja, zgodnie z którą wiersz ma długość maksymalnie 79 znaków.

Jeżeli chcesz skorzystać z tej funkcji w VS Code, wybierz opcję menu *Code/Preferences*, a następnie kliknij *Settings*. W wyświetlnym oknie dialogowym wpisz *rulers*. Zobaczysz ustawienia dla *Editor: Rulers*. Kliknij łącze zatytuowane *Edit in settings.json*. W wyświetlnym pliku musisz dodać następujące ustawienie *editor.rulers*:

*Plik settings.json:*

```
"editor.rulers": [  
  80,  
]
```

To ustawienie spowoduje, że VS Code umieści pionową linię na 80. kolumnie znaków. Można mieć więcej niż jedną pionową linię. Na przykład jeśli chcesz mieć dodatkową linię na 120. znaku, wartość omawianego ustawienia powinna wynosić [80, 120]. Jeżeli nie widzisz pionowych linii, upewnij się, czy masz zapisany plik ustawień. W niektórych systemach być może trzeba również zamknąć i ponownie uruchomić VS Code, aby zmiany zostały wprowadzone.

## Upraszczanie danych wyjściowych

Domyślnie VS Code wyświetla dane wyjściowe programów w osadzonym oknie terminala. Te dane wyjściowe obejmują polecenia użyte do uruchomienia plików. W wielu przypadkach to będzie idealne rozwiązanie, natomiast podczas poznawania Pythona może być rozpraszające.

W celu uproszczenia danych wyjściowych zamknij wszystkie karty otwarte w VS Code, a następnie zamknij VS Code. Uruchom ponownie VS Code i otwórz katalog zawierający pliki Pythona, nad którymi pracujesz. Powinien to być katalog *projekty\_python*, w którym wcześniej zapisałś plik *hello\_world.py*.

Kliknij ikonę *Run/Debug* (przypomina mały trójkąt) z robakiem, a następnie kliknij *Create a launch.json File*. W wyświetlnym oknie dialogowym wybierz opcje Pythona. W edytorze pojawi się plik *launch.json*, w którym należy wprowadzić następującą zmianę:

*Plik launch.json:*

```
{  
  --cięcie--  
  "configurations": [  
    {  
      --cięcie--  
      "console": "internalConsole",
```

```
        "justMyCode": true
    }
}
```

---

To spowoduje zmianę wartości opcji `console` z `integratedTerminal` na `internal` → `Console`. Po zapisaniu pliku otwórz plik `.py`, np. `hello_world.py`, i uruchom go przez naciśnięcie klawiszy `Ctrl+F5`. W panelu danych wyjściowych VS Code kliknij `Debug Console`, jeśli nie została wcześniej wybrana. Zobaczysz jedynie dane wyjściowe programu, które będą odświeżane po każdym jego uruchomieniu.

**UWAGA** *Okno Debug Console jest tylko do odczytu. Nie będzie działało z plikami zawierającymi wywołanie funkcji `input()`, z której zaczeliśmy korzystać w rozdziale 7. Gdy zachodzi potrzeba uruchomienia programu zawierającego tę funkcję, możesz zmienić ustawienie opcji `console` z powrotem na domyślną `integratedTerminal` albo uruchomić go w oddzielnym oknie terminala, jak to wyjaśniłem w rozdziale 1.*

## Dalsze konfigurowanie edytora Sublime Text

VS Code można dostosować na wiele sposobów, aby przygotować jak najefektywniejsze środowisko pracy. Aby rozpocząć poznawanie dostępnych opcji pozwalających dostosować VS Code do własnych potrzeb, z menu `Code` wybierz opcję `Preferences`, a później kliknij `Settings`. Zobaczysz listę zatytułowaną `Commonly Used`. Kliknij dowolny z jej podnagłówków, a zobaczysz kolejne sposoby, w jakie można modyfikować instalację VS Code. Poświęć nieco czasu na sprawdzenie, czy możesz usprawnić sposób działania VS Code. Jednak postaraj się nie koncentrować wyłącznie na konfigurowaniu edytora i nie odkładaj na bok poznawania Pythona.

## Wybrane skróty klawiszowe VS Code

Wszystkie edytory i środowiska IDE oferują efektywne sposoby wykonywania zadań, które każdy musi realizować podczas tworzenia kodu i jego późniejszej obsługi technicznej. Na przykład można bardzo łatwo zastosować wcięcie dla pojedynczego wiersza kodu bądź całego bloku. Równie łatwo można blok wierszy przenieść w górę bądź w dół pliku.

Istnieje zbyt wiele skrótów klawiszowych, aby w tym miejscu wymienić je wszystkie. Dlatego wspomnę jedynie o kilku wybranych, które prawdopodobnie uznasz za użyteczne podczas tworzenia swoich pierwszych plików Pythona. Jeżeli będziesz korzystać z innego edytora tekstu niż VS Code, zapoznaj się ze sposobami efektywnego wykonywania tych samych zadań w tym edytorze.

## Wcięcia i brak wcięć bloków kodu

Aby zastosować wcięcia dla bloku kodu, należy go zaznaczyć i nacisnąć klawisze `Ctrl+J` (`Command+J` w systemie macOS). Natomiast aby zmniejszyć wcięcia bloku kodu, należy nacisnąć klawisze `Ctrl+[` (`Command+[` w systemie macOS).

## Umieszczanie bloku kodu w komentarzu

Aby tymczasowo wyłączyć blok kodu, możesz go zaznaczyć i umieścić w komentarzu, a ten kod zostanie zignorowany przez Pythona. Zaznacz blok kodu i naciśnij klawisze *Ctrl+/* (*Command+/* w systemie macOS). Wiersze w zaznaczonym bloku kodu zostaną poprzedzone znakiem *#* i wcięte na tym samym poziomie, na którym znajduje się kod, aby wskazać, że nie są zwykłymi komentarzami. By wyciągnąć blok kodu z komentarza, zaznacz blok kodu i użyj tego samego polecenia.

## Przenoszenie wierszy w góre bądź w dół

Gdy programy staną się bardziej skomplikowane, być może zacznie pojawiać się potrzeba przenoszenia wierszy kodu w górę bądź w dół pliku. W tym celu zaznacz kod, który chcesz przenieść, i naciśnij klawisze *Alt+strzałka w góre* (*Option+strzałka w góre* w systemie macOS). Ten sam skrót, ale ze strzałką w dół powoduje przeniesienie zaznaczonego bloku w dół pliku.

Jeżeli przenosisz w góre bądź w dół pojedynczy wiersz kodu, możesz kliknąć dowolne miejsce w tym wierszu. Nie musisz zaznaczać całego wiersza, aby móc go przenieść.

## Ukrywanie eksploratora plików

Zintegrowany eksplorator plików w VS Code jest naprawdę wygodny. Jednak czasami może rozpraszać podczas tworzenia kodu bądź zabierać cenne miejsce na mniejszym ekranie. Skrót *Ctrl+B* (*Command+B* w systemie macOS) powoduje wyświetlenie lub ukrycie eksploratora plików.

## Wyszukiwanie kolejnych skrótów klawiszowych

Gdy poznajesz sposoby pracy z kodem, spróbuj dostrzec zadania wykonywane wielokrotnie. Dla każdej czynności wykonywanej w edytorze tekstu prawdopodobnie istnieje skrót. Jeżeli klikasz elementy menu w celu wykonywania zadań edycyjnych, poszukaj dla nich skrótu. Jeżeli często przechodzisz między klawiaturą i myszą, poszukaj skrótu nawigacyjnego, dzięki któremu nie trzeba będzie często sięgać po mysz.

Wszystkie skróty klawiszowe w VS Code można poznać po wybraniu opcji menu *Code/Preferences*, a następnie *Keyboard Shortcuts*. Paska wyszukiwania można użyć do znalezienia konkretnego skrótu. Ewentualnie można przewinąć listę i znaleźć odpowiedni skrót, który pomoże w efektywniejszej pracy.

Pamiętaj, że lepiej jest skoncentrować się na kodzie, nad którym pracujesz, i starać się nie poświęcać zbyt wiele czasu wykorzystywaniem narzędziom.

# Inne edytory tekstu i środowiska IDE

Przekonasz się, że wielu innych programistów korzysta z odmiennych edytorów tekstu. Większość tych programów można skonfigurować w taki sposób, aby stały się równie efektywne jak VS Code. W tym podrozdziale wymieniłem jedynie kilka innych edytorów tekstu, które możesz spotkać.

## IDLE

*IDLE* to domyślny edytor tekstu instalowany wraz z Pythonem. Pracuje się z nim nieco mniej intuicyjnie niż w przypadku innych i znacznie nowocześniejszych edytorów, ale odwołania do niego będziesz spotykał w różnych samouczkach oraz w innych opracowaniach przeznaczonych dla początkujących programistów. Dlatego też warto go wypróbować.

## Geany

*Geany* to prosty edytor tekstu, który wyświetla wszystkie dane wyjściowe w oddzielnym oknie terminala, co pomaga programistom nabyć doświadczenia w pracy z powłoką. Wprawdzie Geany charakteryzuje się bardzo prostym interfejsem użytkownika, ale jednocześnie oferuje na tyle potężne możliwości, że korzysta z niego wielu doświadczonych programistów.

Jeżeli uznasz, że edytor VS Code jest zbyt rozpraszający i oferuje zbyt wiele funkcjonalności, rozważ sięgnięcie po Geany.

## Sublime Text

*Sublime Text* to minimalistyczny edytor tekstu, którego użycie możesz rozważyć, jeśli uznasz VS Code za zbyt rozbudowane narzędzie. Sublime Text oferuje naprawdę przejrzysty interfejs i świetnie sobie radzi podczas pracy z ogólnymi plikami. Jest to edytor, który pozwala skoncentrować się na tworzonym kodzie.

Sublime Text oferuje nieograniczoną czasowo wersję próbную, choć nie jest to oprogramowanie bezpłatne ani otwartoźródłowe. Jeżeli polubisz ten edytor i możesz sobie pozwolić na zakup licencji, zrób to. Zakup jest jednorazowy, to nie jest subskrypcja.

## Emacs i vim

*Emacs* i *vim* to dwa popularne edytory tekstu, bardzo lubiane przez wielu doświadczonych programistów, ponieważ można ich używać praktycznie bez odrywania rąk od klawiatury. Dzięki temu kiedy opanuje się sposób pracy z wymienionymi edytorami, można niezwykle wydajnie zajmować się tworzeniem, czytaniem i modyfikowaniem kodu źródłowego. Niestety, to oznacza również, że nauka edytorów emacs i vim nie należy do prostych zadań. Vim znajduje się w systemie macOS oraz w większości dystrybucji systemu Linux, a zarówno Emacs, jak i Vim mogą działać całkowicie w powłoce. Te edytory są więc często stosowane do tworzenia kodu w serwerach za pomocą zdalnych sesji powłoki.

Programiści bardzo często zachęcają do wypróbowania tych edytorów, choć jednocześnie wielu zaawansowanych programistów zapomina, jak dużo nowych i trudnych rzeczy musi nauczyć się początkujący programista. Dlatego też warto, abyś wiedział o istnieniu tych edytorów, jednak radziłbym Ci wstrzymać się z ich użyciem do czasu, aż poczujesz się pewnie w tworzeniu kodu źródłowego i pracowaniu z nim w znacznie prostszym edytorze tekstu. W ten sposób będziesz mógł skoncentrować się bardziej na nauce programowania niż na używaniu edytora.

## PyCharm

*PyCharm* to popularne wśród programistów Pythona środowisko IDE, ponieważ zostało opracowane specjalnie do tworzenia kodu źródłowego w Pythonie. Wprawdzie pełna wersja wymaga płatnej subskrypcji, ale jest dostępna również bezpłatna — PyCharm Community Edition — którą wielu programistów uznaje za niezwykle użyteczną.

Jeżeli wypróbowujesz PyCharm, to musisz wiedzieć, że domyślnie konfiguruje on odizolowane środowisko dla poszczególnych projektów. Przeważnie jest to dobre rozwiązanie, choć może prowadzić do nieoczekiwanej sposobu działania, jeśli nie rozumiesz tej funkcjonalności.

## Notatniki Jupyter Notebooks

*Notatniki Jupyter Notebooks* tu zupełnie inny rodzaj narzędzia niż typowy edytor tekstu lub środowisko IDE — jest to aplikacja internetowa zbudowana z bloków. Każdy blok składa się z tekstu lub kodu źródłowego. Bloki tekstu są generowane w formacie Markdown, więc można w nich stosować proste formatowanie.

Notatniki Jupyter Notebooks opracowano z myślą o zastosowaniu Pythona w aplikacjach naukowych. Jednak ten projekt został rozbudowany na tyle, że często go stosować także w wielu innych dziedzinach. Gdy pracujesz z notatnikiem Jupyter Notebooks, to zamiast umieszczać komentarze wewnętrz plików kodu, `.py`, tworzysz je w oddzielnych komórkach notatnika. Te komentarze mają postać zwykłego tekstu i można w nich stosować proste formatowanie, takie jak nagłówki, listy wypunktowane i łącza między poszczególnymi sekcjami kodu. Każdy blok może być uruchomiony niezależnie, co pozwala testować niewielkie fragmenty programu lub uruchamiać wszystkie jednocześnie. Każdy blok ma własny obszar przeznaczony na dane wyjściowe, a pomiędzy poszczególnymi obszarami możesz dowolnie przechodzić wedle potrzeb.

Czasami podejście zastosowane w notatnikach Jupyter Notebooks może wydawać się kłopotliwe ze względu na relacje zachodzące między poszczególnymi komórkami. Po zdefiniowaniu funkcji w jednej komórce jest ona dostępna także w innych. W większości sytuacji jest to oczekiwane, choć jednocześnie może być źródłem pomyłek w większych notatnikach lub jeśli nie do końca rozumiesz zasadę działania środowiska notatników Jupyter Notebooks.

Jeżeli w Pythonie wykonujesz jakiekolwiek zadania naukowe bądź związane z danymi, w końcu niemal na pewno zetniesz się z Jupyter Notebooks.

# C

## Uzyskiwanie pomocy



KAŻDY NA PEWNYM ETAPIE NAUKI PROGRAMOWANIA DOJDZIE DO MOMENTU, W KTÓRYM NAPOTKA PROBLEM UNIEMOŻLIWIJĄCY MU PÓJŚCIE DALEJ. JEDNĄ Z NAJWAŻNIEJSZYCH UMIEJĘTNOŚCI, JAKĄ POWINIEN OPA nować programista, jest efektywne rozwiązywanie problemów. W tym dodatku przedstawię kilka sposobów radzenia sobie w sytuacjach, gdy programowanie sprawia trudności.

### Pierwsze kroki

Jeśli utknąłeś w martwym punkcie, najpierw powinieneś ocenić sytuację. Zanim poprosisz o jakąkolwiek pomoc, spróbuj odpowiedzieć sobie na trzy poniższe pytania:

- Co próbujesz zrobić?
- Co udało Ci się zrobić do tej pory?
- Jakich wyników oczekujesz?

Twoje odpowiedzi powinny być jak najbardziej konkretne. Jeśli chodzi o pierwsze pytanie, to odpowiedź typu „Próbuje zainstalować najnowszą wersję Pythona w moim nowym laptopie z Windowsem” jest wystarczająco szczegółowa, aby inni członkowie społeczności Pythona mogli Ci pomóc. Natomiast odpowiedź w stylu: „Próbuje zainstalować Pythona”, zawiera zbyt mało informacji, aby można było zaaproponować pytającemu konkretną pomoc.

Odpowiedź na drugie pytanie powinna być jak najbardziej szczegółowa, ponieważ w ten sposób unikniesz sugestii wykonywania kroków, które już wcześniej podjąłeś. Dlatego też odpowiedź typu: „Wszedłem na stronę <http://python.org/downloads/> i kliknąłem przycisk *Download*. Po pobraniu instalatora uruchomiłem go”, jest dużo bardziej pomocna niż zdanie w stylu: „Wszedłem na stronę Pythona i pobrałem jakiś plik”.

Jeśli chodzi o odpowiedź na ostatnie pytanie, to szukając w internecie rozwiązania problemu lub prosząc o pomoc, dobrze mieć zanotowaną dokładną treść otrzymanego komunikatu błędu.

Czasami udzielenie sobie odpowiedzi na te trzy pytania pozwala spojrzeć na problem z innej perspektywy i dostrzec przeoczony wcześniej szczegół, który uniemożliwiał pójście dalej. Programiści nawet mają odpowiednie określenie na tego rodzaju sytuację: *debugowanie metodą gumowej kaczuszki*. Jeśli swoją sytuację wyjaśnisz jasno gumowej kaczce (lub jakiemukolwiek innemu przedmiotowi) i zadasz konkretne pytanie, często będziesz w stanie znaleźć odpowiedź na to pytanie. Niektóre sklepy ze sprzętem komputerowym mają w asortymencie prawdziwe gumowe kaczki właśnie po to, aby zachęcić programistów do prowadzenia „dyskusji z gumową kaczką”.

## Spróbuj jeszcze raz

Cofnięcie się do samego początku i ponowne wykonanie danej czynności czasem pomaga rozwiązać wiele problemów. Przykładem może być próba utworzenia pętli `for` w programie przedstawionym w tej książce. Być może pominąłeś jakiś prosty element, na przykład dwukropkę na końcu wiersza zawierającego polecenie `for`. Ponowne wykonanie wszystkich czynności może pomóc uniknąć popełnienia tego samego błędu.

## Chwila odpoczynku

Jeśli pracujesz nad jakimś problemem już od dłuższego czasu, to krótka przerwa na odpoczynek będzie jednym z najlepszych możliwych ruchów, i jest to podejście zdecydowanie warte wypróbowania. Kiedy dość długo pracujesz na pewnym zadaniu, mózg zaczyna koncentrować się tylko na jednym rozwiązaniu. Tracisz perspektywę i nie potrafisz właściwie spojrzeć na przyjęte założenia. W takim przypadku przerwa pomaga przywrócić świeże spojrzenie na problem. To nie musi być dłuża przerwa — wystarczy zająć się czymś innym, tak aby pozwolić odpocząć umysłowi. Jeśli siedzisz przez dłuższy czas przed komputerem, postaw na zajęcia fizyczne. To może być krótki spacer lub po prostu wyjście z domu. Inna możliwość to napicie się wody lub zjedzenie czegoś lekkiego i zdrowego.

Jeśli jesteś już naprawdę sfrustrowany i zniechęcony, warto odłożyć pracę nad rozwiązaniem problemu nawet na następny dzień. Zdrowy sen i nocny odpoczynek niemal zawsze owocują następnego dnia dobrymi pomysłami i rozwiązaniami.

## Korzystaj z zasobów tej książki

Materiały przygotowane do tej książki są dostępne na stronie [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/). Znajdziesz w nich wiele pomocnych sekcji dotyczących konfiguracji używanego systemu oraz pracy z kodem przedstawionym w poszczególnych rozdziałach. Jeśli do tej pory tego nie zrobileś, przyjrzyj się tym zasobom i sprawdź, czy znajdziesz w nich pomocne materiały.

## Wyszukiwanie informacji w internecie

Istnieje duże prawdopodobieństwo, że ktoś wcześniej spotkał się już z tym samym problemem co Ty i umieścił o tym informację w internecie. Umiejętność dobrego wyszukiwania informacji i zadawania właściwych pytań z pewnością pomoże w znalezieniu zasobów, które okazać się mogą pomocne podczas rozwiązywania problemu. Na przykład jeśli usiłujesz zainstalować najnowszą wersję Pythona w nowym systemie Windows, wpisanie w ulubionej wyszukiwarce internetowej wyrażenia „instalacja python windows” i ograniczenie wyników do zasobów z ostatniego roku może naprowadzić na właściwą odpowiedź.

Również wyszukiwanie z podaniem dokładnego komunikatu błędu okazuje się wyjątkowo pomocne. Przyjmijmy na przykład założenie, że kiedy próbujesz uruchomić sesję Pythona w wierszu poleceń systemu Windows, otrzymujesz następujący komunikat błędu:

---

```
> python hello_world.py
Nazwa 'python' nie jest rozpoznawana jako polecenie wewnętrzne lub zewnętrzne,
↪program wykonywalny lub plik wsadowy
>
```

---

Wyszukanie pełnego komunikatu błędu (tutaj *Nazwa 'python' nie jest rozpoznawana jako polecenie wewnętrzne lub zewnętrzne, program wykonywalny lub plik wsadowy*) prawdopodobnie zaowocuje znalezieniem jakiejś dobrej rady.

Kiedy zaczniesz wyszukiwać tematy związane z programowaniem, natychmiast pojawi się wiele odnośników prowadzących do wielu stron. Niektóre z nich opiszę pokrótko, abyś wiedział, jak bardzo mogą okazać się pomocne.

## Stack Overflow

*Stack Overflow* (<http://stackoverflow.com/>) to jeden z najbardziej popularnych serwisów zawierających pytania i udzielone na nie odpowiedzi. Jest skierowany do programistów i zwykle w wynikach wyszukiwania dotyczących Pythona pojawia się na pierwszej stronie. Idea tego serwisu polega na tym, że użytkownicy zadają pytania dotyczące problemu, którego nie potrafią rozwiązać, a inni członkowie grupy próbują udzielić pomocnych odpowiedzi. Użytkownicy mogą głosować na najbardziej pomocne odpowiedzi, dlatego najlepsze rozwiązania znajdują się zwykle na początku.

Na wiele podstawowych pytań dotyczących Pythona znajdziesz w serwisie Stack Overflow bardzo jasne odpowiedzi, ponieważ z czasem społeczność je dopracowała i udoskonalila. Ponadto użytkownicy są zachęcani do aktualizowania postów, więc udzielone odpowiedzi z reguły pozostają aktualne. W trakcie powstawania tej książki w serwisie Stack Overflow znajdowało się prawie dwa miliony pytań związanych z Pythonem, na które udzielono odpowiedzi.

Istnieje pewna zasada, o której należy wiedzieć przed zamieszczeniem pytania w serwisie Stack Overflow. Otóż pytanie powinno być krótkim przykładem problemu, z którym się zetknąłeś. Jeżeli zamieścisz 5–20 wierszy kodu generujących błędów, z którym masz problem, i jeśli odpowiedziałeś sobie na pytania zamieszczone na początku tego dodatku, to masz szansę otrzymać pomoc. Natomiast jeśli udostępnisz projekt z wieloma ogromnymi plikami, prawdopodobnie tej pomocy nie uzyskasz. Na stronie <https://stackoverflow.com/help/how-to-ask> znajduje się świetne wyjaśnienie, jak należy zadawać dobre pytania. Zamieszczone tam sugestie mają zastosowanie podczas szukania pomocy w każdej społeczności programistów.

## Oficjalna dokumentacja Pythona

Oficjalna dokumentacja Pythona (<https://docs.python.org/>) jest w przypadku początkujących programistów strzałem na chybiił trafl, ponieważ jej celem jest bardziej dokumentowanie samego języka niż skupianie się na jego objaśnianiu. Przykłady zaprezentowane w oficjalnej dokumentacji powinny działać, ale niekoniecznie będziesz mógł zrozumieć wszystko, co zostało w nich pokazane. Mimo tego to wciąż bardzo dobre źródło wiedzy, które warto sprawdzić, gdy odnośniki do niego pojawią się w wynikach wyszukiwania. Sama dokumentacja z całą pewnością stanie się cenniejsza dla Ciebie, gdy będziesz kontynuował poznawanie programowania w języku Python.

## Oficjalna dokumentacja biblioteki

Jeśli używasz określonej biblioteki, takiej jak Pygame, matplotlib czy Django, bardzo pomocne okażą się prowadzące do oficjalnej dokumentacji łącza, które pojawią się w wynikach wyszukiwania. Przykładem witryny zawierającej taką dokumentację może być <http://docs.djangoproject.com/>. Jeśli planujesz wykorzystanie do wolnej z wymienionych bibliotek, na pewno doskonałym pomysłem będzie zapoznanie się z jej oficjalną dokumentacją.

## r/learnpython

Serwis internetowy Reddit składa się z wielu subforum zwanych *subreddits*. Subreddit *r/learnpython* (<http://reddit.com/r/learnpython>) jest dość aktywne i pomocne. Znajdziesz tutaj pytania zadane przez innych użytkowników, a także możesz zadać własne. Często zyskasz wiele perspektyw dotyczących zadawanego pytania, co może okazać się naprawdę pomocne w zrozumieniu zagadnienia, z którego opowiadaniem próbujesz sobie poradzić.

## Posty na blogach

Wielu programistów prowadzi blog i dzieli się z czytelnikami informacjami dotyczącymi tego, nad czym aktualnie pracują. Zwróć uwagę na daty postów, aby ustalić, w jakim stopniu te informacje mogą być przydatne w przypadku używanej przez Ciebie wersji Pythona.

## Discord

*Discord* to środowisko czatu internetowego dla społeczności Pythona, w którym można szukać pomocy na tematy związane z Pythonem.

Jeżeli chcesz wypróbować Discord, zacznij od odwiedzenia witryny <https://pythondiscord.com/> i kliknięcia łącza *Discord* w prawym górnym rogu. Jeżeli masz już konto Discord, możesz się zalogować za jego pomocą. Natomiast jeśli jeszcze nie masz konta, podaj nazwę użytkownika i wykonuj polecenia wyświetlane na ekranie, aby w ten sposób dokończyć rejestrację w serwisie Discord.

Jeśli odwiedzasz Python Discord po raz pierwszy, najpierw musisz zaakceptować reguły społeczności, a dopiero później możesz w niej w pełni uczestniczyć. Akceptacja reguł społeczności umożliwia przyłączenie się do dowolnego interesującego Cię kanału. Gdy szukasz pomocy, upewnij się, że post został umieszczony na kanale *Python Help*.

## Slack

*Slack* to kolejne środowisko czatu internetowego. Bardzo często jest wykorzystywany do wewnętrznej komunikacji w firmie, choć istnieje również wiele grup publicznych, do których można dołączyć. Jeżeli chcesz poszukać grup związanych z Pythonem, zacznij od witryny internetowej <https://pyslackers.com/web>. Kliknij łącze *Slack* u góry strony, wpisz adres e-mail i poczekaj na zaproszenie.

Gdy znajdziesz się już w obszarze *Python Developers*, zobaczysz listę kanałów. Kliknij *Channels*, a następnie wybierz interesujący Cię temat. Na początek warto rozważyć kanały *#learning\_python* i *#django*.

# D

## Używanie Gita do kontroli wersji



OPROGRAMOWANIE SŁUŻĄCE DO KONTROLI WERSJI POZWALA NA UTWORZENIE MIGAWKI PROJEKTU W CHWILI, GDY DZIAŁA PRAWIDŁOWO I ZGODNIE Z OCZEKIWANAMI. DZIĘKI TEMU PO WPROWADZENIU ZMIAN W PROJEKcie, na przykład w postaci implementacji nowej funkcji, masz możliwość cofnięcia tych zmian i przywrócenia aplikacji do poprzedniego stanu, jeśli aktualnie nie działa zgodnie z oczekiwaniemi.

Używanie oprogramowania przeznaczonego do kontroli wersji daje większą swobodę i elastyczność podczas wprowadzania usprawnień oraz pozwala na popelnianie błędów bez obaw o zniszczenie projektu. Wprawdzie to ma bardzo istotne znaczenie przede wszystkim w ogromnych projektach, ale może okazać się przydatne również w mniejszych, nawet jeśli pracujesz nad programem mieszącym się w pojedynczym pliku.

W tym dodatku dowiesz się, jak zainstalować oprogramowanie Git oraz wykorzystać je do kontroli wersji programów, nad którymi pracujesz. Obecnie *Git* to najpopularniejszy system przeznaczony do kontroli wersji. Wiele jego zaawansowanych funkcji ułatwia zespołom programistów współpracę przy ogromnych projektach. Warto w tym miejscu dodać, że podstawowe funkcje systemu Git doskonale sprawdzają się także w indywidualnej pracy programistów nad projektem. Git implementuje system kontroli wersji, monitorując zmiany wprowadzane we wszystkich plikach projektu. Jeżeli popełnisz błąd, możesz po prostu powrócić do poprzednio zachowanego stanu.

# Instalacja Gita

Wprawdzie oprogramowanie Git działa we wszystkich najważniejszych systemach operacyjnych, ale jego instalacja przebiega odmiennie w poszczególnych systemach. W poniższych sekcjach przedstawiłem informacje dotyczące instalacji oprogramowania Git w różnych systemach operacyjnych.

Oprogramowanie Git jest standardowo dołączane do niektórych systemów i często łączone z innymi pakietami, które być może już wcześniej zainstalowałeś. Zanim spróbujesz zainstalować Gita, sprawdź, czy nie znajduje się już w Twoim systemie. Otwórz nowe okno powłoki i wydaj polecenie `git --version`. Jeśli zostaną wygenerowane dane wyjściowe zawierające informacje o konkretnej wersji, oznacza to, że Git jest już zainstalowany w Twoim systemie. Natomiast w przypadku komunikatu zachęcającego do instalacji lub aktualizacji Gita, po prostu kieruj się podawanymi wskazówkami.

Jeżeli nie widzisz żadnych wskazówek na ekranie i używasz systemu Windows bądź macOS, program instalacyjny Gita możesz pobrać z witryny <https://git-scm.com/>. Natomiast jeśli jesteś użytkownikiem Linuksa wykorzystującego menedżer pakietów apt, to aby zainstalować Gita, wydaj polecenie `sudo apt install git`.

## Konfiguracja Gita

Git rejestruje informacje o tym, kto dokonuje zmian w projekcie, nawet jeśli nad danym projektem pracuje tylko jedna osoba. W tym celu system Git musi znać Twoją nazwę użytkownika i adres e-mail. Musisz więc podać nazwę użytkownika, natomiast niekoniecznie musisz podawać używany przez Ciebie adres e-mail:

---

```
$ git config --global user.name "nazwa_użytkownika"  
$ git config --global user.email "użytkownik@example.com"
```

---

Jeśli zapomnisz o podaniu tych informacji, Git zapyta o nie podczas pierwszego zatwierdzania zmian w repozytorium.

Dobrym podejściem będzie również zdefiniowanie domyślnej nazwy dla gałęzi głównej w każdym projekcie. Dobrą nazwą dla takiej gałęzi jest `main`.

---

```
$ git config --global init.defaultBranch main
```

---

To ustawienie konfiguracyjne oznacza, że w każdym nowym projekcie zarządzanym przez Gita na początku będzie znajdowała się gałąź `main`, do której będzie domyślnie przekazywany kod.

# Tworzenie projektu

Przygotujemy teraz projekt do pracy. W tym celu utwórz w systemie katalog o nazwie *git\_cwiczenia*. Następnie umieść w nim plik *hello\_git.py* zawierający prosty program w Pythonie.

*Plik hello\_git.py:*

---

```
print("Witaj, świecie Gita!")
```

---

Tego programu będziemy używać do poznania podstawowych funkcjonalności systemu Git.

## Ignorowanie plików

Pliki z rozszerzeniem *.pyc* są automatycznie generowane na podstawie plików *.py*, więc nie ma potrzeby ich monitorowania przez Git. Te pliki są przechowywane w katalogu o nazwie *\_pycache\_*. Aby system Git zignorował ten katalog, utwórz plik specjalny o nazwie *.gitignore* — z kropką na początku nazwy pliku i bez rozszerzenia pliku — a następnie dodaj do niego poniższy wiersz kodu.

*Plik .gitignore:*

---

```
_pycache_/
```

---

W ten sposób nakazujemy Gitowi ignorowanie wszystkich plików znajdujących się w katalogu *\_pycache\_*. Dzięki użyciu pliku *.gitignore* unikamy zaśmiecania projektu i ułatwiamy sobie pracę.

Może być potrzebne zmodyfikowanie ustawień używanego edytora tekstu w taki sposób, aby pokazywał również ukryte pliki (czyli pliki o nazwie rozpoczynającej się od kropki). W systemie Windows przejdź do *Eksploratora plików*, kliknij kartę *Widok*, a następnie upewnij się, że jest zaznaczona opcja *Ukryte elementy*. W systemie macOS naciśnij klawisze *Command+Shift+. (kropka)* w dowolnym oknie Findera. W systemie Linux poszukaj opcji zatytułowanej *Ukryte pliki*.

**UWAGA** Jeżeli używasz systemu macOS, do pliku *.gitignore* dodaj jeszcze *.DS\_Store*. To są pliki ukryte przechowujące informacje o ustawieniach poszczególnych katalogów w systemie macOS i zaśmiecają Ci projekt, jeśli nie dodasz ich do pliku *.gitignore*.

# Inicjalizacja repozytorium

Skoro masz już katalog zawierający plik programu napisanego w Pythonie i plik `.gitignore`, możesz zainicjować repozytorium Gita. W tym celu przejdź do powłoki, a następnie do katalogu `git_cwiczenia` i wydaj poniższe polecenie:

---

```
git_cwiczenia$ git init
Initialized empty Git repository in git_cwiczenia/.git/
git_cwiczenia$
```

---

Dane wyjściowe pokazują, że system Git zainicjował puste repozytorium w katalogu `git_cwiczenia`. *Repozytorium* to zbiór plików w programie, który jest aktywnie monitorowany przez Gita. Wszystkie pliki wykorzystywane przez Gita do zarządzania repozytorium są przechowywane w ukrytym katalogu o nazwie `.git`, z którym jednak w ogóle nie musisz pracować. Mimo to nie usuwaj tego katalogu, ponieważ stracisz wtedy całą historię projektu.

## Sprawdzanie stanu

Zanim zrobisz cokolwiek innego, sprawdź aktualny stan projektu, jak pokazałem poniżej:

---

```
git_cwiczenia$ git status
On branch main ❶
  No commits yet

Untracked files: ❷
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    hello_git.py

nothing added to commit but untracked files present (use "git add" to track) ❸
git_cwiczenia$
```

---

W systemie Git słowo *branch* (*gałęź*) przedstawia wersję projektu, nad którą pracujesz. W omawianym przykładzie wyraźnie widać, że pracujemy w gałęzi o nazwie `main` (patrz wiersz ❶). Za każdym razem, gdy sprawdzisz stan projektu, powinieneś otrzymywać informacje, że pracujesz w gałęzi `main`. Następnie mamy informacje o przeprowadzeniu początkowego zatwierdzenia. Wspomniane *zatwierdzenie* (*commit*) to migawka projektu w określonym punkcie w czasie.

Git informuje nas o niemonitorowanych plikach znajdujących się w projekcie (patrz wiersz ❷), ponieważ jeszcze nie wskazaliśmy plików, które mają być monitorowane. W wierszu ❸ znajduje się informacja o niedodaniu żadnych plików do aktualnego zatwierdzenia, choć istnieją niemonitorowane pliki, które być może będziesz chciał umieścić w repozytorium ❸.

# Dodawanie plików do repozytorium

Dodamy teraz dwa pliki do repozytorium i ponownie sprawdzimy jego stan:

```
git_cwiczenia$ git add . ①
git_cwiczenia$ git status ②
On branch main
Initial commit

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file:  .gitignore ③
  new file:  hello_git.py

git_cwiczenia$
```

Polecenie `git add .` sprawia, że do repozytorium zostają dodane wszystkie pliki projektu, które nie są jeszcze monitorowane (patrz wiersz ①), o ile nie zostały wymienione w pliku `.gitignore`. To nie powoduje zatwierdzenia tych plików, a jedynie informuje Gita o konieczności zwrócenia na nie uwagi. Jeśli teraz ponownie sprawdzimy stan projektu, to okaże się, że system Git wykrywa pewne zmiany wymagające zatwierdzenia (patrz wiersz ②). Etykieta *new file* oznacza nowo dodane pliki do repozytorium (patrz wiersz ③).

## Zatwierdzanie plików

Przeprowadzimy teraz pierwszą operację zatwierdzenia plików:

```
git_cwiczenia$ git commit -m "Rozpoczęcie projektu." ①
[main (root-commit) cea13dd] Rozpoczęcie projektu. ②
 2 files changed, 5 insertion(+) ③
  create mode 100644 .gitignore
  create mode 100644 hello_git.py
git_cwiczenia$ git status ④
On branch main
nothing to commit, working directory clean
git_cwiczenia$
```

Na początku wydaliśmy polecenie `git commit -m "Rozpoczęcie projektu"` (patrz wiersz ①), aby utworzyć migawkę aktualnego stanu projektu. Opcja `-m` nakazuje systemowi Git zarejestrować komunikat ujęty w cudzysłów i umieścić go w dzienniku projektu. Wygenerowane dane wyjściowe pokazują, że pracujemy w gałęzi `main` (patrz wiersz ②) i zmodyfikowane zostały dwa pliki (patrz wiersz ③).

Jeżeli teraz sprawdzimy stan projektu, wyraźnie widać, że znajdujemy się w gałęzi `main`, a katalog roboczy jest uznawany za czysty (patrz wiersz ④). Tego

rodzaju komunikat będziesz chciał widzieć za każdym razem, gdy będziesz przeprowadzać operację zatwierdzenia projektu, który w swoim aktualnym stanie działa poprawnie. Jeśli otrzymasz inny komunikat, powinieneś bardzo dokładnie go przeczytać. Istnieje spore prawdopodobieństwo, że zapomniałeś dodać plik, zanim rozpoczęłeś operację zatwierdzania.

## Sprawdzanie dziennika projektu

Git przechowuje dziennik wszystkich operacji zatwierdzenia przeprowadzonych w projekcie. Teraz zajrzymy do tego dziennika:

```
git_cwiczenia$ git log  
commit cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main)  
Author: eric <eric@example.com>  
Date:   Mon Jun 6 19:37:26 2022 -0800
```

```
Rozpoczęcie projektu.  
git_cwiczenia$
```

W trakcie każdej operacji zatwierdzenia Git generuje unikatowy 40-znakowy identyfikator. W dzienniku zapisywane są informacje o tym, kto przeprowadził daną operację zatwierdzenia, kiedy została przeprowadzona ta operacja (data) oraz jaki był komunikat podany przez użytkownika. Nie zawsze potrzebne są te wszystkie informacje, więc Git oferuje możliwość wyświetlenia zawartości dziennika w znacznie prostszej postaci:

```
git_cwiczenia$ git log --pretty=oneline  
cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Rozpoczęcie projektu.  
git_cwiczenia$
```

Opcja `--pretty=oneline` powoduje wyświetlenie dwóch najważniejszych fragmentów informacji, czyli identyfikatora zatwierdzenia oraz komunikatu podanego przez użytkownika w trakcie przeprowadzania operacji zatwierdzenia.

## Drugie zatwierdzenie

Aby przekonać się o prawdziwej potędze systemu kontroli wersji, musimy dokonać zmiany w projekcie, a następnie ją zatwierdzić. W omawianym przykładzie po prostu dodajemy następny wiersz kodu w pliku `hello_git.py`.

## Plik hello\_git.py:

---

```
print("Witaj, świecie Gita!")
print("Dzień dobry wszystkim!")
```

---

Jeśli teraz sprawdzimy stan projektu, to zobaczymy, że system Git odnotował dokonaną zmianę w pliku:

```
git_cwiczenia$ git status
On branch main ①
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   hello_git.py ②

no changes added to commit (use "git add" and/or "git commit -a") ③
git_cwiczenia$
```

---

W wierszu ① wyświetlona jest nazwa gałęzi, w której obecnie pracujemy. Wiersz ② zawiera nazwę zmodyfikowanego pliku. Natomiast w wierszu ③ mamy komunikat informujący, że żadna zmiana nie została jeszcze zatwierdzona. Przystępujemy więc do zatwierdzenia zmiany i ponownego sprawdzenia stanu projektu:

```
git_cwiczenia$ git commit -am "Rozbudowa powitania." ①
[main 945fa13] Rozbudowa powitania.
  1 file changed, 1 insertion(+), 1 deletion(-)
git_cwiczenia$ git status ②
On branch main
nothing to commit, working directory clean
git_cwiczenia$ git log --pretty=oneline ③
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Rozbudowa powitania.
ceal3ddc51b885d05a410201a54faf20e0d2e246 Rozpoczęcie projektu.
git_cwiczenia$
```

---

Przeprowadzamy nową operację zatwierdzenia. W poleceniu `git commit` użyliśmy opcji `-am` (patrz wiersz ①). Opcja `-a` informuje system Git o konieczności uwzględnienia w przeprowadzanej operacji zatwierdzenia wszystkich plików zmodyfikowanych w repozytorium. (Gdy między następującymi po sobie zatwierdzeniami utworzysz jakikolwiek nowy plik, wówczas aby dodać nowe pliki do repozytorium, po prostu ponownie wydaj polecenie `git add .`). Natomiast opcja `-m` nakazuje systemowi Git umieścić w dzienniku komunikat podany w trakcie przeprowadzanej operacji zatwierdzania.

Kiedy ponownie sprawdzimy stan projektu, to zobaczymy, że katalog roboczy znów jest uznawany za czysty (patrz wiersz ②). Na końcu przeglądamy dwie operacje zatwierdzenia, o których informacje znajdują się w dzienniku (patrz wiersz ③).

# Przywracanie stanu projektu

Teraz zobaczysz, w jaki sposób można cofnąć zmiany i przywrócić projekt do poprzedniego stanu. Najpierw dodamy nowy wiersz kodu do pliku `hello_git.py`.

*Plik hello\_git.py:*

---

```
print("Witaj, świecie Gita!")
print("Dzień dobry wszystkim!")

print("O nie, zepsułem projekt!")
```

---

Zapisz plik i uruchom ten program.

Sprawdź stan projektu i przekonaj się, że Git zanotował tę zmianę.

---

```
git_cwiczenia$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   hello_git.py ①

no changes added to commit (use "git add" and/or "git commit -a")
git_cwiczenia$
```

---

Git wykrywa zmianę dokonaną w pliku `hello_git.py` (patrz wiersz ①) — w tym momencie można przeprowadzić operację zatwierdzenia, jeśli zachodzi taka potrzeba. Jednak tym razem zamiast zatwierdzić zmiany, chcemy przywrócić projekt do poprzedniego stanu, w którym działał on bez zarzutu. Nie będziemy podejmować żadnych działań w pliku `hello_git.py` — nie usuniemy wiersza kodu i nie użyjemy funkcji `Cofnij` oferowanej przez edytor tekstu. Zamiast tego w sesji powłoki wydaj poniższe polecenia:

---

```
git_cwiczenia$ git restore .
git_cwiczenia$ git status
On branch main
nothing to commit, working directory clean
git_cwiczenia$
```

---

Polecenie `git restore nazwa_pliku` pozwala odrzucić wszystkie zmiany w podanym pliku wprowadzone od chwili poprzedniej operacji zatwierdzenia. Z kolei polecenie `git restore .` odrzuca wszelkie zmiany we wszystkich plikach wprowadzone od ostatniej operacji zatwierdzenia i przywraca projekt do stanu, w którym znajdował się po poprzednim zatwierdzeniu.

Kiedy powrócisz do edytora tekstu, zobaczysz, że plik `hello_git.py` został przywrócony do następującej postaci:

---

```
print("Witaj, świecie Gita!")
print("Dzień dobry wszystkim!")
```

---

O ile powrót do poprzedniego stanu w tym prostym przykładzie może wydawać się trywialny, to tego rodzaju operacja nie będzie już taka łatwa w przypadku pracy nad ogromnymi projektami z dziesiątkami zmodyfikowanych plików. Do poprzedniego działającego stanu przywrócone będą wszystkie pliki, które zostały zmienione od chwili ostatniej operacji zatwierdzenia. Ta funkcja jest niezwykle użyteczna, ponieważ pozwala na wprowadzenie dowolnej ilości zmian w projekcie podczas implementacji nowej funkcjonalności. Następnie jeśli nowa funkcja nie działa zgodnie z oczekiwaniemi, zmiany możesz cofnąć bez żadnej szkody dla projektu. Nie musisz pamiętać tych zmian i wycofywać ich ręcznie. System Git zrobi to za Ciebie.

**UWAGA** *Być może trzeba będzie kliknąć w oknie edytora tekstu, aby odświeżyć plik i wyświetlić jego poprzednią wersję.*

## Przywracanie projektu do poprzedniego stanu

Istnieje możliwość przywrócenia projektu do dowolnego poprzedniego stanu zarchiwowanego przez przeprowadzenie operacji zatwierdzenia. W tym celu należy użyć polecenia `checkout` i podać sześć pierwszych znaków identyfikatora stanu, do którego chcesz przywrócić projekt. Przywracając projekt do wybranego stanu, możesz przejrzeć wcześniejsze operacje zatwierdzenia. Następnie możesz powrócić do ostatniego zatwierdzenia, lub też odrzucić ostatnie zmiany i podjąć pracę nad projektem od stanu, w jakim znajdował się jeszcze wcześniej:

---

```
git_cwiczenia$ git log --pretty=oneline
945fa13af128a266d0114eebb7a3276f7d58ecd2 (HEAD -> main) Rozbudowa powitania.
cea13ddc51b885d05a410201a54faf20e0d2e246 Rozpoczęcie projektu.
git_cwiczenia$ git checkout cea13d
Note: checking out 'cea13d'.
```

You are in 'detached HEAD' state. You can look around, make experimental ❶ changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command again. Example:

```
git switch -c <new_branch_name>
```

Or undo this operation with: ②

```
git switch -
```

Turn off this advice by setting config variable advice.detachedHead to false

```
HEAD is now at cea13d Rozpoczęcie projektu.  
git_cwiczenia$
```

---

Kiedy przywracasz projekt do wcześniejszego stanu, opuszczasz gałąź *main* i przechodzisz do stanu określonego przez Git mianem *detached HEAD* (patrz wiersz ①). Obecnym stanem projektu jest *HEAD*, natomiast słowo *detached* oznacza opuszczenie nazwanej gałęzi (w omawianym przykładzie to gałąź *main*).

Jeżeli chcesz powrócić do gałęzi *main*, musisz zastosować się do przedstawionej sugestii (patrz wiersz ②) i cofnąć poprzednią operację:

```
git_cwiczenia$ git switch -  
Previous HEAD position was cea13d Rozpoczęcie projektu.  
Switched to branch 'main'  
git_cwiczenia$
```

---

W ten sposób powracasz do gałęzi *main*. O ile nie zamierzasz wykorzystywać pewnych bardziej zaawansowanych funkcji systemu Git, najlepszym rozwiązaniem będzie unikanie wprowadzania jakichkolwiek zmian w projekcie po przywróceniu go do stanu, w jakim znajdował się podczas wcześniejszego zatwierdzenia. Jeżeli jednak jesteś jedyną osobą pracującą nad danym projektem, chcesz odrzucić ostatnie zmiany oraz powrócić do wcześniejszego stanu, możesz wyzrobić projekt do wybranego wcześniejszego stanu. W tym celu pracując w gałęzi *main*, należy wydać następujące polecenia:

```
git_cwiczenia$ git status ①  
On branch main  
nothing to commit, working directory clean  
git_cwiczenia$ git log --pretty=oneline ②  
945fa13af128a266d0114eabb7a3276f7d58ecd2 (HEAD -> main) Rozbudowa  
↳powitania.  
cea13ddc51b885d05a410201a54faf20e0d2e246 Rozpoczęcie projektu.  
git_cwiczenia$ git reset --hard cea13d ③  
HEAD is now at cea13d Rozpoczęcie projektu.  
git_cwiczenia$ git status ④  
On branch main  
nothing to commit, working directory clean  
git_cwiczenia$ git log --pretty=oneline ⑤  
cea13ddc51b885d05a410201a54faf20e0d2e246 (HEAD -> main) Rozpoczęcie projektu.  
git_cwiczenia$
```

---

Zaczynamy od sprawdzenia stanu repozytorium, aby mieć pewność, że znajdujemy się w gałęzi *main* (patrz wiersz ①). Następnie sprawdzamy dziennik i widzimy obie operacje zatwierdzenia (patrz wiersz ②). Wydajemy polecenie `git reset --hard` wraz z pierwszymi sześcioma znakami identyfikatora stanu, do którego chcemy trwale przywrócić projekt (patrz wiersz ③). Ponownie sprawdzamy stan i potwierdzamy to, że znajdujemy się w gałęzi *main* i nie ma żadnych zmian do zatwierdzenia (patrz wiersz ④). Po spojrzeniu do dziennika widzimy, że znajdujemy się w stanie, do którego chcieliśmy przywrócić projekt (patrz wiersz ⑤).

## Usuwanie repozytorium

Czasem może się zdarzyć, że popsujesz coś w historii repozytorium i nie wiesz, jak z tego wybrnąć. Jeśli znajdziesz się w takiej sytuacji, najpierw spróbuj poprosić o pomoc przy użyciu metod omówionych w dodatku C. Natomiast jeśli nie potrafisz naprawić repozytorium, a pracujesz sam nad projektem, możesz kontynuować pracę z plikami i pozbyć się historii projektu, usuwając katalog *.git*. To nie będzie miało wpływu na aktualny stan jakiegokolwiek pliku, choć spowoduje usunięcie wszystkich operacji zatwierdzenia. Dlatego też nie będziesz mógł przywrócić projektu do żadnego wcześniejszego stanu.

W tym celu przejdź do menedżera plików i usuń repozytorium *.git*, lub też usuń ten katalog z poziomu powłoki. Innymi słowy pracę z repozytorium będziesz musiał rozpocząć od początku, aby zapewnić monitorowanie plików. Poniżej przedstawiłem cały proces przeprowadzony w sesji powłoki:

---

```
git_cwiczenia$ git status ①
On branch main
nothing to commit, working directory clean
git_cwiczenia$ rm -rf .git ②
git_cwiczenia$ git status ③
fatal: Not a git repository (or any of the parent directories): .git
git_cwiczenia$ git init ④
Initialized empty Git repository in git_cwiczenia/.git/
git_cwiczenia$ git status ⑤
On branch main
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    hello_git.py

nothing added to commit but untracked files present (use "git add" to track)
git_cwiczenia$ git add . ⑥
git_cwiczenia$ git commit -m "Rozpoczęcie od początku."
[main (root-commit) 14ed9db] Rozpoczęcie od początku.
  2 files changed, 5 insertions(+)
```

```
create mode 100644 .gitignore
create mode 100644 hello_git.py
git_cwiczenia$ git status ⑦
On branch main
nothing to commit, working directory clean
git_cwiczenia$
```

---

Najpierw sprawdzamy stan i widzimy, że katalog roboczy jest uznawany za czysty (patrz wiersz ①). Następnie używamy polecenia `rm -rf .git` (del .git w systemie Windows) w celu usunięcia katalogu `.git` (patrz wiersz ②). Kiedy sprawdzamy stan po usunięciu katalogu `.git`, wyświetlony komunikat informuje nas, że to nie jest repozytorium Gita (patrz wiersz ③). Wszystkie dane używane przez Gita do monitorowania repozytorium są przechowywane w katalogu `.git`, więc jego usunięcie powoduje usunięcie całego repozytorium.

Za pomocą polecenia `git init` możemy od początku utworzyć repozytorium (patrz wiersz ④). Sprawdzenie stanu wskazuje, że ponownie jesteśmy na etapie początkowym, oczekując na pierwsze zatwierdzenie (patrz wiersz ⑤). Dodajemy pliki do repozytorium i przeprowadzamy pierwszą operację zatwierdzenia (patrz wiersz ⑥). Ponowne sprawdzenie stanu wskazuje, że znajdujemy się w nowej gałęzi `main` i nie ma żadnych zmian do zatwierdzenia (patrz wiersz ⑦).

Wprawdzie efektywne użycie systemu kontroli wersji wymaga pewnej wprawy, ale kiedy zaczniesz już z niego korzystać, nie będziesz potrafił się bez tego obejść.

# E

## Rozwiązywanie problemów z wdrożeniami



WDRAŻANIE APLIKACJI JEST OGROMNIE SATYSFAKCJONUJĄCE, GDY PRZEBIEGA ZGODNIE Z OCZEKIWANIAMI, ZWŁASZCZA JEŚLI JESZCZE NIGDY NIE MIELIŚMY OKAZJI PRZEPROWADZAĆ TAKIEJ OPERACJI. JEDNAK ISTNIEJE WIELE PROBLEMÓW, KTÓRE MOGĄ ZAKŁÓCIĆ PROCES WDRAŻANIA. NIESTETY CZĘŚĆ Z NICH MOŻE OKAĆ SIĘ TRUDNA DO ZNALEZIENIA I USUNIĘCIA. W TYM DODATKU POSTARAM SIĘ POMÓC CI ZROZUMIEĆ NOWOCZESNE PODĘJCIA DO WDRAŻANIA I PRZESTAWIĘ KONKRETNE SPOSOBY ROZWIĄZYWAŃIA PROBLEMÓW POJAWIAJĄCYCH W TRAKCIE PROCESU WDRAŻANIA APLIKACJI, GDY NIE WSZYSTKO DZIAŁA ZGODNIE Z OCZEKIWANIAMI.

Jeżeli zamieszczone tutaj informacje dodatkowe okażą się niewystarczające do pomyślnego wdrożenia, odwiedź stronę [https://ehmatthes.github.io/pcc\\_3e/](https://ehmatthes.github.io/pcc_3e/), na której znajdziesz pomocne zasoby. Znajdujące się tam informacje niemal na pewno pomogą Ci w udanym przeprowadzeniu wdrożenia.

### Jak wygląda proces wdrażania?

Kiedy próbujesz rozwiązywać problemy napotykane podczas wdrażania, dobrze jest wcześniej wiedzieć, jak przebiega typowy proces wdrażania. Wdrożenie to proces, w wyniku którego projekt działający lokalnie zostaje skopiowany na zdalny serwer, aby mógł obsługiwać żądania pochodzące od użytkowników w internecie. Zdalne środowisko różni się od typowego systemu lokalnego pod wieloma

ważnymi względami — użytkownik prawdopodobnie nie używa tego samego systemu operacyjnego (OS) co programista, a ponadto pojedynczy serwer fizyczny zwykle ma uruchomionych wiele serwerów wirtualnych.

Podczas wdrażania projektu, czyli *przekazywania* go na zdalny serwer, należy wykonać następujące kroki:

- Utworzenie serwera wirtualnego w fizycznym komputerze znajdującym się w centrum danych.
- Nawiązanie połączenia między systemem lokalnym i zdalnym serwerem.
- Skopiowanie plików projektu na zdalny serwer.
- Ustalenie wszystkich zależności projektu i ich zainstalowanie w zdalnym serwerze.
- Skonfigurowanie bazy danych i przeprowadzenie wszelkich istniejących migracji.
- Skopiowanie plików statycznych (arkusze stylów CSS, pliki JavaScriptu i pliki multimedialne) do miejsca, z którego będą mogły być efektywnie udostępniane.
- Uruchomienie nowego serwera w celu rozpoczęcia obsługi żądań przychodzących.
- Rozpoczęcie routingu żądań przychodzących do projektu, gdy stanie się on gotowy na ich obsługę.

Gdy spojrzesz na listę wszystkich kwestii, którymi trzeba się zająć podczas wdrożenia, stwierdzisz, że to nie dziwnego, iż ta operacja często kończy się niepowodzeniem. Na szczęście gdy już dowiesz się, o co w niej chodzi, masz większe szanse na ustalenie, w czym tak naprawdę tkwi problem. Jeżeli jesteś w stanie znaleźć źródło niepowodzenia procesu wdrożenia, zyskujesz możliwość usunięcia problemu i masz szansę, że kolejna próba wdrożenia zakończy się sukcesem.

Projekt możesz tworzyć lokalnie w komputerze działającym pod kontrolą jednego systemu operacyjnego, a następnie przekazywać go na serwer, w którym jest uruchomiony zupełnie inny system operacyjny. Trzeba koniecznie wiedzieć, do jakiego systemu operacyjnego będzie przekazywany projekt, ponieważ w ten sposób można sobie znacznie ułatwić pracę i uniknąć pewnych problemów. W chwili powstawania tej książki podstawowy zdalny serwer w usłudze Platform.sh działał pod kontrolą systemu operacyjnego Debian Linux. Warto w tym miejscu dodać, że w większości zdalnych serwerów działa system z rodziny Linux.

## Podstawy rozwiązywania problemów

Niektóre kroki procesu rozwiązywania problemów są charakterystyczne dla poszczególnych systemów operacyjnych, ale do tej kwestii wkrótce powróć. Przede wszystkim warto poznać kroki, które należy wykonać podczas rozwiązywania problemów napotykanych w trakcie wdrażania.

Twoim najlepszym zasobem będą dane wyjściowe wygenerowane podczas próby przekazania projektu do zdalnego serwera. Te dane mogą wyglądać przerażająco. Jeżeli dopiero zaczynasz zajmować się wdrażaniem aplikacji, dane te mogą wydawać się ściśle techniczne i zwykle takie są. Dobrą wiadomością jest to, że nie musisz rozumieć wszystkiego w tego rodzaju danych wyjściowych. Podczas przeglądania danych wyjściowych dzienników zdarzeń masz dwa cele: zidentyfikowanie kroków wdrożenia zakończonych pomyślnie i ustalenie kroków, które zakończyły się niepowodzeniem. Jeżeli jesteś w stanie to zrobić, masz szansę ustalić, co należy zmienić w projekcie bądź procesie wdrożenia, aby następna próba zakończyła się sukcesem.

## Kierowanie się podpowiedziami wyświetlanymi na ekranie

Czasami platforma docelowa będzie generowała komunikaty zawierające wyraźną sugestię, jak rozwiązać problem. Na przykład w przedstawionym tutaj fragmencie kodu znajduje się komunikat wygenerowany w trakcie próby utworzenia projektu Platform.sh przed inicjalizacją repozytorium Git, a następnie została przeprowadzona próba przekazania projektu do zdalnego serwera:

---

```
$ platform push
Enter a number to choose a project: ①
[0] 11_project (votohz4451jyg)
> 0

[RootNotFoundException] ②
Project root not found. This can only be run from inside a project directory.

To set the project for this Git repository, run: platform project:set-remote [id] ③
```

---

Mamy tutaj próbę przekazania projektu, przy czym projekt lokalny nie został jeszcze powiązany z projektem zdalnym. Dlatego działające w powloce narzędzie Platform.sh prosi o wskazanie zdalnego projektu, do którego ma zostać przekazany lokalny ①. W omawianym przykładzie została podana wartość 0, aby w ten sposób wybrać jedyny dostępny projekt. Jednak następnie widzimy wyjątek RootNotFoundException ②. Tak się zdarzyło, ponieważ podczas analizy projektu lokalnego Platform.sh szuka katalogu `.git`, aby ustalić, w jaki sposób nawiązać połączenie projektu lokalnego ze zdalnym. Ponieważ w omawianym przykładzie nie istniał katalog `.git` podczas tworzenia zdalnego projektu, odpowiednie połączenie nigdy nie zostało nawiązane. Działające w powloce narzędzie Platform.sh proponuje rozwiązanie tego problemu ③. Mianowicie informuje o możliwości wskazania zdalnego projektu, który powinien zostać powiązany z lokalnym. W tym celu należy skorzystać z polecenia `project:set-remote`.

Warto wypróbować tę sugestię:

---

```
$ platform project:set-remote votohz4451jyg  
Setting the remote project for this repository to: ll_project (votohz4451jyg)
```

```
The remote project for this repository is  
now set to: ll_project (votohz4451jyg)
```

---

W poprzednich danych wyjściowych narzędzie działające w powłoce wyświetliło identyfikator zdalnego projektu, votohz4451jyg. Dlatego po wydaniu zasugerowanego polecenia i użyciu podanego identyfikatora narzędzie było w stanie nawiązać połączenie między projektami lokalnym i zdalnym.

Spróbujmy raz jeszcze przekazać projekt do zdalnego serwera:

---

```
$ platform push  
Are you sure you want to push to the main (production) branch? [Y/n] y Pushing  
→HEAD to the existing environment main  
--cięcie-
```

---

Tym razem operacja zakończyła się pomyślnie i wykorzystanie podpowiedzi wyświetlonych na ekranie było strzałem w dziesiątkę.

Zachowaj ostrożność podczas wydawania poleceń, których sposób działania nie do końca rozumiesz. Jeżeli jednak masz powody wierzyć, że dane polecenie jest nieszkodliwe, i jeśli ufasz źródłu rekomendującemu to polecenie, wówczas może być sensowne wypróbowanie sugestii przedstawionej przez używane narzędzie.

**UWAGA** *Pamiętaj, że istnieją osoby, które będą Cię nakłaniały do wydawania poleceń powodujących na przykład usunięcie zawartości dysku komputera bądź narażających system na ryzyko zdalnego ataku. Zastosowanie się do sugestii narzędzia dostarczanego przez zaufaną firmę bądź organizację to zupełnie co innego niż stosowanie się do sugestii przypadkowo napotkanych osób. Zachowaj ogromną ostrożność za każdym razem, gdy masz do czynienia ze zdalnymi połączeniami.*

## Odczytywanie danych wyjściowych dzienników zdarzeń

Jak wcześniej wspomniałem, dane wyjściowe generowane po wykonaniu polecenia takiego jak `platform push` mogą zawierać wiele informacji i jednocześnie być przytaczające. Przejrzyj przedstawiony tutaj fragment danych wyjściowych wygenerowanych po wydaniu polecenia `platform push` i zobacz, czy jesteś w stanie dostrzec problem:

```
--cięcie--  
Collecting soupsieve==2.3.2.post1  
  Using cached soupsieve-2.3.2.post1-py3-none-any.whl (37 kB)  
Collecting sqlparse==0.4.2  
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)  
Installing collected packages: platformshconfig, sqlparse,...  
Successfully installed Django-4.1 asgiref-3.5.2 beautifulsoup4-4.11.1...  
W: ERROR: Could not find a version that satisfies the requirement gunicorn  
W: ERROR: No matching distribution found for gunicorn  
  
130 static files copied to '/app/static'.  
  
Executing pre-flight checks...  
--cięcie--
```

---

Gdy próba wdrożenia zakończy się niepowodzeniem, dobrą strategią będzie przejrzenie danych wyjściowych w celu sprawdzenia, czy coś wygląda jak komunikat ostrzeżenia lub błędu. Ostrzeżenia są dość powszechnne, to często komunikaty informujące o przyszłych zmianach w zależnościach projektu i mają pomóc programistom w rozwiązywaniu potencjalnych problemów, zanim faktycznie doprowadzą one do błędów.

Zakończone powodzeniem przekazanie projektu na zdalny serwer może prowadzić do wygenerowania komunikatów ostrzeżeń, ale nie błędów. W przedstawionym przykładzie Platform.sh nie potrafi zainstalować wymaganego modułu gunicorn. Mamy tutaj do czynienia z błędem w pliku *requirements\_remote.txt*, w którym powinna znajdować się nazwa modułu gunicorn (zapis z użyciem jednej litery *R*). Nie zawsze łatwo będzie wychwycić w wygenerowanych danych wyjściowych główną przyczynę błędu, zwłaszcza jeśli powoduje on wygenerowanie kaskadowej ilości komunikatów ostrzeżeń i błędów. Podobnie jak w przypadku analizowania stosu wywołań w systemie lokalnym, także podczas analizowania danych wyjściowych dziennik zdarzeń dobrze jest dokładnie przejrzeć kilka pierwszych i kilka ostatnich błędów. Większość błędów między nimi będzie miała postać komunikatów wygenerowanych przez pakiety wewnętrzne, informujących o pewnych problemach i przekazujących te informacje do innych pakietów wewnętrznych. Rzeczywisty błąd, który należy usunąć, zazwyczaj będzie wymieniony jako jeden z pierwszych bądź ostatnich.

Czasami będziesz w stanie dostrzec problem, w innych zaś przypadkach nie będziesz wiedzieć, o co chodzi w wygenerowanych danych wyjściowych. Zdecydowanie warto spróbować, a pomyślne zdiagnozowanie problemu na podstawie dziennika zdarzeń przynosi ogromną satysfakcję. Gdy będziesz poświęcać coraz więcej czasu na przeglądanie danych wyjściowych dzienników zdarzeń, naukujesz się lepiej wychwytywać informacje o największym znaczeniu dla Ciebie.

# Rozwiązywanie problemów dotyczących konkretnego systemu operacyjnego

Projekt możesz tworzyć, korzystając z dowolnego systemu operacyjnego, a następnie przekazywać go do innego dowolnego hosta. Narzędzia przeznaczone do przekazywania projektów zostały opracowane w taki sposób, że potrafią zmodyfikować projekt tak, aby można go było z powodzeniem uruchomić w zdalnym systemie. Jednak mogą się pojawić pewne problemy ściśle związane z danym systemem operacyjnym.

W trakcie procesu wdrażania Platform.sh jedno z najczęstszych źródeł problemu jest związane z zainstalowaniem narzędzia działającego w powłoce. Spójrz na polecenie, które trzeba w tym celu wydać:

---

```
$ curl -fsS https://platform.sh/cli/installer | php
```

---

Na początku mamy wywołanie `curl`, czyli narzędzia umożliwiającego wykonywanie żądań do zdalnych zasobów. To polecenie zostało tutaj użyte w powłoce w celu wykonania żądania do określonego adresu URL. W efekcie z serwera Platform.sh zostanie pobrany program instalacyjny CLI. Argument w postaci `-fsS` zawiera opcje zmieniające sposób działania polecenia `curl`. Opcja `-f` nakazuje zawieszenie wyświetlania większości komunikatów błędów, aby działający w powłoce program instalacyjny mógł je obsługiwać zamiast zgłaszać użytkownikowi. Opcja `-S` nakazuje „ciche” działanie, pozwala programowi instalacyjnemu decydować, które informacje zostaną wyświetcone w powłoce. Z kolei opcja `-S` nakazuje poleceniu `curl` wyświetlenie komunikatu błędu, jeżeli wykonanie całego polecenia zakończy się niepowodzeniem. Zapis `| php` na końcu polecenia nakazuje systemowi uruchomienie pobranego programu instalacyjnego za pomocą interpretera PHP, ponieważ działające w powłoce narzędzie Platform.sh zostało utworzone właśnie w PHP.

Dlatego w celu zainstalowania wspomnianego narzędzia system wymaga polecenia `curl` i interpretera PHP. Do pracy z narzędziem powłoki Platform.sh potrzebujesz także Gita i możliwości wydawania poleceń powłoki Bash. *Bash* to rodzaj języka dostępnego w większości środowisk serwerowych. Większość nowoczesnych systemów ma mnóstwo miejsca na zainstalowanie takich narzędzi.

W kolejnych punktach postaram się wyjaśnić kwestie związane z wymaganiami dla systemu operacyjnego. Jeżeli jeszczе nie masz zainstalowanego Gita, zapoznaj się z zamieszczonymi w dodatku D informacjami na temat instalacji Gita w Twoim systemie operacyjnym.

**UWAGA** *Doskonale narzędzie pomocne w zrozumieniu poleceń powłoki jest dostępne pod adresem <https://explainshell.com/>. Wpisz nazwę polecenia, które próbujesz zrozumieć, a ta witryna wyświetli dokumentację przeznaczoną dla poszczególnych fragmentów tego polecenia. Wypróbuj ją z poleceniem użyтыm do zainstalowania działającego w powłoce narzędzia Platform.sh.*

## **Wdrażanie z poziomu systemu Windows**

W ostatnich latach można zauważać ponowny wzrost popularności systemu Windows wśród programistów. Windows ma zintegrowanych wiele różnych elementów innych systemów operacyjnych, co zapewnia użytkownikom mnóstwo opcji w zakresie lokalnego tworzenia oprogramowania oraz pracy ze zdalnymi systemami.

Jedną z największych trudności związanych z przeprowadzaniem wdrożenia z poziomu Windowsa jest to, że jądro tego systemu operacyjnego nie jest takie samo jak używane przez zdalne serwery oparte na Linuksie. Podstawowy system Windows ma odmienny zestaw narzędzi i języków programowania niż bazowy system Linux. Dlatego przekazanie projektu programistycznego z Windowsa wymaga wyboru rozwiązania w zakresie integracji w środowisku lokalnym zbioru narzędzi Linuksa.

## **Windows Subsystem for Linux**

Jednym z popularnych podejść jest użycie *Windows Subsystem for Linux* (WSL), czyli środowiska umożliwiającego działanie Linuksa bezpośrednio w Windowsie. Jeżeli masz skonfigurowane środowisko WSL, to używanie w Windowsie działającego w powłoce narzędzia Platform.sh staje się równie łatwe jak korzystanie z tego narzędzia w Linuksie. Narzędzie Platform.sh nie będzie wiedziało, że zostało uruchomione w Windowsie; ze swojej perspektywy będzie działało w środowisku Linuksa.

Przygotowanie środowiska WSL to proces składający się z dwóch kroków. Pierwszy to zainstalowanie WSL, natomiast drugi to wybór dystrybucji Linuksa przeznaczonej do zainstalowania w WSL. Ta operacja będzie nieco bardziej złożona, niż tutaj wskazałem. Jeżeli interesuje Cię to podejście i jeszcze nie masz skonfigurowanego środowiska WSL, zapoznaj się z dokumentacją dostępną na stronie <https://learn.microsoft.com/en-us/windows/wsl/about>. Po przygotowaniu WSL możesz skorzystać z informacji zamieszczonych w części tego dodatku poświęconej Linuksowi.

## **Git Bash**

Inne podejście w zakresie budowania lokalnego środowiska programistycznego, z którego będzie można przeprowadzać wdrożenia, polega na użyciu *Git Bash*, czyli środowiska zgodnego z powłoką Bash, ale działającego w Windowsie. Git Bash otrzymujesz w przypadku instalacji Gita za pomocą programu instalacyjnego pobranego z witryny <https://git-scm.com/>. Wprawdzie takie podejście działa, ale nie jest tak dopracowane jak WSL. W tym przypadku trzeba będzie używać terminala Windowsa do wykonywania określonych kroków i terminala Git Bash do wykonywania innych.

Przede wszystkim trzeba zainstalować PHP. Do tego można wykorzystać XAMPP, czyli pakiet łączący PHP z kilkoma innymi narzędziami dla programistów. Przejdz do witryny <https://www.apachefriends.org/> i kliknij odpowiedni przycisk, aby pobrać XAMPP w wersji dla Windowsa. Uruchom ten program instalacyjny.

Zobaczysz ostrzeżenie dotyczące kontroli konta użytkownika (UAC). Kliknij przycisk *OK*. Zaakceptuj wszystkie ustawienia domyślne programu instalacyjnego.

Gdy program instalacyjny zakończy działanie, musisz dodać PHP do ścieżki systemowej. Dzięki temu Windows będzie wiedział, gdzie szukać pliku wykonywanego, gdy zechcesz uruchomić PHP. W menu *Start* wpisz *zmienne* i wybierz *Edytuj zmienne środowiskowe systemu*. Kliknij przycisk *Zmienne środowiskowe....* Wybierz zmienną *Path*, a następnie kliknij przycisk *Edytuj....* Potem kliknij przycisk *Nowy* w celu dodania nowej ścieżki dostępu do bieżącej listy ścieżek dostępu. Zakładając, że zostały użyte ustawienia domyślne podczas instalacji XAMPP, w wyświetlnym oknie dialogowym dodaj *C:\xampp\php* i kliknij *OK*. Po zakończeniu zamknij wszystkie otwarte systemowe okna dialogowe.

Po spełnieniu niezbędnych wymagań można przystąpić do instalacji działającego w powłoce narzędzia Platform.sh. Konieczne będzie użycie terminala Windowsa z uprawnieniami administracyjnymi. W menu *Start* wpisz *cmd*, a następnie prawym przyciskiem myszy kliknij element *Wiersz poleceń* i wybierz opcję *Uruchom jako administrator*. W wyświetlnym oknie terminala wydaj następujące polecenie:

---

```
> curl -fsS https://platform.sh/cli/installer | php
```

---

To spowoduje zainstalowanie działającego w powłoce narzędzia Platform.sh, zgodnie z przedstawionymi wcześniej informacjami.

Będziesz również pracować w terminalu Git Bash. Aby go uruchomić, przejdź do menu *Start* i wyszukaj *git bash*. Kliknij aplikację *Git Bash* w wynikach wyszukiwania, a zobaczysz na ekranie okno terminala. Możesz w nim używać tradycyjnych poleceń powłoki Linuksa, takich jak *ls*, a także poleceń Windowsa, takich jak *dir*. Aby upewnić się, że instalacja zakończyła się pomyślnie, wydaj polecenie *platform list*. Powinieneś otrzymać listę wszystkich poleceń oferowanych przez narzędzie Platform.sh CLI. Od tej chwili wdrożenia przeprowadzasz za pomocą narzędzia Platform.sh CLI uruchomionego w oknie terminala Git Bash.

## Wdrażanie z poziomu systemu macOS

Wprawdzie system operacyjny macOS nie bazuje na Linuksie, ale oba pochodzą z tej samej rodziny i wykorzystują podobną filozofię działania. Dlatego wiele poleceń i sposobów działania znanych z systemu macOS można używać także w środowisku zdalnego serwera. Być może trzeba będzie zainstalować zasoby przeznaczone dla programistów, aby mieć te narzędzia dostępne w lokalnym środowisku macOS. Jeżeli w trakcie pracy zobaczysz okno dialogowe z pytaniem dotyczącym instalacji *narzędzi programistycznych działających w powłoce*, kliknij przycisk *Zainstaluj*, aby zostały zainstalowane w systemie.

Podczas instalacji działającego w powłoce narzędzia Platform.sh prawdopodobnie trzeba będzie się upewnić, że dostępny jest interpreter PHP. Jeżeli otrzymasz komunikat błędu informujący o braku polecenia *php*, musisz zainstalować

interpreter PHP. Jednym z najłatwiejszych sposobów instalacji PHP jest użycie menedżera pakietów Homebrew, który ułatwia instalację wielu różnych pakietów niezbędnych programistom. Jeżeli jeszcze nie masz zainstalowanego tego menedżera, odwiedź witrynę <https://brew.sh/> i wykonaj zamieszczone tam polecenia.

Po zainstalowaniu Homebrew następujące polecenie pozwala zainstalować interpreter PHP w systemie macOS:

---

```
$ brew install php
```

---

Operacja może chwilę potrwać, ale po jej zakończeniu będzie można przeprowadzić zakończony pomyślnie proces instalacji działającego w powłoce narzędzia Platform.sh.

## Wdrażanie z poziomu systemu Linux

Skoro większość środowisk serwerowych bazuje na Linuksie, raczej nie będziesz mieć trudności podezas instalacji działającego w powłoce narzędzia Platform.sh i jego późniejszego używania. Jeżeli spróbujesz zainstalować to narzędzie w nowym systemie Ubuntu, otrzymasz komunikat wskazujący, jakie dokładnie pakiety są niezbędne:

---

```
$ curl -fsS https://platform.sh/cli/installer | php
Command 'curl' not found, but can be installed with:
sudo apt install curl
Command 'php' not found, but can be installed with:
sudo apt install php-cli
```

---

Faktycznie wygenerowane dane wyjściowe będą zawierały nieco więcej informacji o jeszcze innych pakietach, a także informacje dotyczące wersji. Następujące polecenie spowoduje zainstalowanie w systemie pakietów curl i php-cli:

---

```
$ sudo apt install curl php-cli
```

---

Po wykonaniu tego polecenia będzie można przeprowadzić zakończony pomyślnie proces instalacji działającego w powłoce narzędzia Platform.sh. Ponieważ środowisko lokalne będzie bardzo podobne do większości środowisk hostinguowych opartych na Linuksie, umiejętności zdobyte podczas pracy w terminalu systemu lokalnego będzie można wykorzystać także w pracy ze zdalnym środowiskiem.

# Inne podejścia w zakresie wdrożenia

Jeżeli serwis Platform.sh Ci nie odpowiada lub zechcesz wypróbować inne podejście, do dyspozycji masz wiele platform hostingowych. Wdrożenie w części z nich będzie przebiegało podobnie do procesu omówionego w rozdziale 20. Z kolei część będzie stosowała odmienne podejście i konieczne może być zastosowanie kroków omówionych na początku dodatku.

- Platform.sh umożliwia użycie przeglądarki WWW w celu wykonywania zadań, do których jest przeznaczone narzędzie działające w powloce. Jeżeli interfejs bazujący na przeglądarce WWW lubisz bardziej niż pracę w powloce, być może będziesz preferować właśnie to podejście.
- Istnieje jeszcze wielu innych dostawców usług hostingowych oferujących możliwość pracy za pomocą zarówno narzędzia działającego w powloce, jak i narzędzia bazującego na przeglądarce WWW. Część z nich oferuje powłokę w przeglądarce WWW, więc nie trzeba będzie niczego instalować w systemie.
- Niektórzy dostawcy umożliwiają przekazanie projektu do zdalnego serwisu hostingu kodu źródłowego, takiego jak np. GitHub, a następnie powiązanie repozytorium GitHub z witryną dostawcy usług hostingu. Host pobiera kod z repozytorium GitHub zamiast wymagać od użytkownika przekazywanego z systemu lokalnego bezpośrednio do hosta. Serwis Platform.sh również obsługuje to rozwiązanie.
- Część dostawców oferuje szeroką gamę usług, spośród których można wybierać, aby w ten sposób przygotować działającą infrastrukturę dla projektu. To zwykle wymaga dokładniejszego zrozumienia procesu wdrożenia i potrzeb zdalnego serwera, aby móc obsłużyć dany projekt. Do takich usług zaliczają się Amazon Web Services (AWS) i Microsoft Azure. Monitorowanie kosztów na tych platformach może być nieco trudniejsze, ponieważ każda usługa może być rozliczana niezależnie od pozostałych.
- Wielu użytkowników umieszcza projekty na wirtualnych serwerach prywatnych (ang. *virtual private server*, VPS). W takim podejściu wypożyczycie się serwer wirtualny, który działa podobnie jak zdalny. Można się do niego zalogować, zainstalować w nim oprogramowanie niezbędne do uruchomienia projektu, skopiować kod źródłowy, zdefiniować odpowiednie uprawnienia i umożliwić serwerowi rozpoczęcie akceptowania żądań pochodzących od użytkowników.

Regularnie pojawiają się nowe platformy hostingowe i podejścia. Znajdź najlepsze dla siebie rozwiązanie i poświęć nieco czasu na poznanie stosowanego w nim procesu wdrażania. Zapewnij przez długi czas obsługę tego projektu, aby wiedzieć, co w rozwiązaniu danego dostawcy usług hostingowych sprawdza się świetnie, a co się nie sprawdziło. Nie istnieje doskonała platforma hostingowa. Trzeba prowadzić ocenę na bieżąco i sprawdzać, czy możliwości oferowane przez danego dostawcę są wystarczająco dobre dla Twojego projektu.

I jeszcze słowo ostrzeżenia dotyczące wyboru platformy wdrożenia i ogólnie podejścia do wdrożenia. Część osób będzie ochoczo próbowała Cię zachęcić do przesadnie skomplikowanego procesu wdrożenia, dzięki któremu Twój projekt stanie się wysoce niezawodny i gotowy do obsługi milionów użytkowników. Wielu programistów poświęca sporo czasu, pieniędzy i energii na tworzenie skomplikowanych strategii wdrażania, a później się okazuje, że mało kto chce używać takiego projektu. Większość projektów Django można umieścić na niewielkich serwerach, a mimo to będą one w stanie obsługiwać dziesiątki tysięcy żądań na minutę. Jeżeli Twój projekt nie osiąga takiego poziomu ruchu sieciowego, skonfiguruj wdrożenie do pracy na minimalnej platformie hostingowej, zanim zdecydujesz się na inwestycję w infrastrukturę przeznaczoną dla największych witryn internetowych na świecie.

Wdrożenie bywa bardzo trudną operacją, ale przynosi ogromną satysfakcję, gdy wdrożony projekt działa zgodnie z oczekiwaniami. Podejmij to wyzwanie i nie obawiaj się poprosić o pomoc, jeśli pojawi się taka potrzeba.

# PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**  
<http://program-partnerski.helion.pl>